NAVAL
POSTGRADUATE
SCHOOL

MONTEREY, CALIFORNIA

THESIS

CHARACTERISTICS OF THE BINARY DECISION DIAGRAMS OF BOOLEAN BENT FUNCTIONS

by

Neil Brendan Schafer

September 2009

Thesis Advisor: Jon T. Butler
Thesis Co-Advisor: Pantelimon Stanica

Approved for public release; distribution is unlimited

REPORT DOCUMENTATION PAGE	*Form Approved OMB No. 0704-0188*

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503.

1. AGENCY USE ONLY *(Leave blank)*	**2. REPORT DATE** September 2009	**3. REPORT TYPE AND DATES COVERED** Master's Thesis

4. TITLE AND SUBTITLE Characteristics of the Binary Decision Diagrams of Boolean Bent Functions	**5. FUNDING NUMBERS**
6. AUTHOR(S) Neil Brendan Schafer	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000	**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING /MONITORING AGENCY NAME(S) AND ADDRESS(ES) N/A	**10. SPONSORING/MONITORING AGENCY REPORT NUMBER**

11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution is unlimited	**12b. DISTRIBUTION CODE** A

13. ABSTRACT (maximum 200 words)

Boolean bent functions have desirable cryptographic properties in that they have maximum nonlinearity, which hardens a cryptographic function against linear cryptanalysis attacks. Furthermore, bent functions are extremely rare and difficult to find. Consequently, little is known generally about the characteristics of bent functions.

One method of representing Boolean functions is with a reduced ordered binary decision diagram. Binary decision diagrams (BDD) represent functions in a tree structure that can be traversed one variable at a time. Some functions show speed gains when represented in this form, and binary decision diagrams are useful in computer aided design and real-time applications.

This thesis investigates the characteristics of bent functions represented as BDDs, with a focus on their complexity. In order to facilitate this, a computer program was designed capable of converting a function's truth table into a minimally realized BDD.

Disjoint quadratic functions (DQF), symmetric bent functions, and homogeneous bent functions of 6-variables were analyzed, and the complexities of the minimum binary decision diagrams of each were discovered. Specifically, DQFs were found to have size $2n – 2$ for functions of n-variables; symmetric bent functions have size $4n – 8$, and all homogeneous bent functions of 6-variables were shown to be P-equivalent.

14. SUBJECT TERMS Binary Decision Diagrams, Boolean Bent Functions, Homogeneous Functions, Disjoint Quadratic Functions, Symmetric Bent Functions, P-Equivalence, Minimization, Hardware Complexity, Circuit Complexity, Graphical Interface, Nonlinearity, Hamming Distance, Cryptography	**15. NUMBER OF PAGES** 175
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT Unclassified	**18. SECURITY CLASSIFICATION OF THIS PAGE** Unclassified	**19. SECURITY CLASSIFICATION OF ABSTRACT** Unclassified	**20. LIMITATION OF ABSTRACT** UU

THIS PAGE INTENTIONALLY LEFT BLANK

CHARACTERISTICS OF THE BINARY DECISION DIAGRAMS OF BOOLEAN BENT FUNCTIONS

Neil Brendan Schafer
Lieutenant, United States Navy
BSCpE, Virginia Tech, 2004

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING

from the

NAVAL POSTGRADUATE SCHOOL
September 2009

Author: Neil Brendan Schafer

Approved by: Jon T. Butler
Thesis Advisor

Pantelimon Stanica
Thesis Co-Advisor

Professor Jeffrey B. Knorr
Chairman, Department of Electrical and Computer Engineering

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

Boolean bent functions have desirable cryptographic properties in that they have maximum nonlinearity, which hardens a cryptographic function against linear cryptanalysis attacks. Furthermore, bent functions are extremely rare and difficult to find. Consequently, little is known generally about the characteristics of bent functions.

One method of representing Boolean functions is with a reduced ordered binary decision diagram. Binary decision diagrams (BDD) represent functions in a tree structure that can be traversed one variable at a time. Some functions show speed gains when represented in this form, and binary decision diagrams are useful in computer aided design and real-time applications.

This thesis investigates the characteristics of bent functions represented as BDDs, with a focus on their complexity. In order to facilitate this, a computer program was designed capable of converting a function’s truth table into a minimally realized BDD.

Disjoint quadratic functions (DQF), symmetric bent functions, and homogeneous bent functions of 6-variables were analyzed, and the complexities of the minimum binary decision diagrams of each were discovered. Specifically, DQFs were found to have size $2n - 2$ for functions of n-variables; symmetric bent functions have size $4n - 8$, and all homogeneous bent functions of 6-variables were shown to be P-equivalent.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF FIGURES

LIST OF TABLES

THIS PAGE INTENTIONALLY LEFT BLANK

EXECUTIVE SUMMARY

Bent functions are Boolean functions that have a maximum Hamming distance from the linear functions and their complements – otherwise known as the affine functions. That is, bent functions have maximum nonlinearity. High nonlinearity is a desirable property for functions used in cryptographic applications because it hardens the cryptographic system from linear cryptanalysis attacks.

Another useful aspect of bent functions in terms of cryptography is that they are rare and difficult to find. While this increases their security, this is also frustrating because this makes them difficult to research. Many known sets of bent function families can be designed generally for a function of *n*-variables, but the vast majority must be found through enumeration of all Boolean functions. For this reason, identifying the characteristics of bent functions from a variety of perspectives may prove valuable in unlocking methods of more efficient discovery.

One method for representing a Boolean function is with a Reduced Ordered Binary Decision Diagram (ROBDD, or BDD, for short). BDDs are tree-like graph structures that provide a condensed form of the function that can be traversed one variable at a time. Complex functions can be calculated fairly quickly with BDDs, which makes them desirable for real-time and computer aided design applications.

It is known that for many human designed Boolean functions, BDDs tend to have simple shapes. This is likely because functions with simple characteristics are useful. Good examples of simple but useful functions are the AND, OR, and XOR functions, which have simple BDDs.

On the other hand, no one has looked closely at the BDDs of known bent functions. Due to their nonlinearity, it is expected bent functions would have high complexity in relation to other human designed functions. Furthermore, little is known about how the shape of bent function BDDs or how the BDDs of different types of bent functions may be related.

This thesis explores the BDDs of bent functions. Since BDD construction can be quite time consuming by hand, the first step was to create a program capable of generating a BDD given the truth table of a Boolean function. Although many programs are available that can convert a function into a BDD data structure, there are none known that integrate a graphical display of the BDD. Creating a program to aid in the visualization necessary was the first order of business.

BDD complexity can also be very sensitive to variable ordering. For instance, one variable ordering of an 8-variable Disjoint Quadratic Function results in a BDD with 14 nodes, while another “diabolical” ordering results in a BDD with 37 nodes. Since in almost all cases, a minimized BDD is desirable, the program had to be able to implement a method for permuting variables in order to find trees of a minimum size.

Designing this program, called BDDViewer, resulted in the realization that the BDD can be fully implemented by parsing the function’s truth table. Specifically, the edges of each node split the truth table in half, with each node representing a sub-function. By viewing each node as a unique sub-function’s truth table, an ROBDD can be constructed with no redundant, or isomorphic sub-graphs.

By implementing the BDD construction in this way, the variable order permutations also had to be properly reflected in the corresponding truth table. The Johnson-Trotter permutation algorithm was used to find all possible variable orderings. However, a new algorithm was developed to manipulate the truth table to reflect the movement of the variables. The details of this algorithm can be seen in Chapter IV.

Once the program was complete, the BDDs of a sampling of known bent functions were investigated. Specifically, disjoint quadratic functions (DQFs), symmetric bent functions, and homogeneous functions of algebraic degree 3 on 6-variables were observed. XORing these functions with linear functions was also performed to identify bent functions’ relationship with their affine classes.

The BDDs of the DQFs and the symmetric bent functions proved to be visually striking. In fact, they both demonstrated clearly recognizable patterns that allowed for the extrapolation of the complexity of such functions generally for *n*-variables. Specifically, the minimum BDDs of DQFs were found to have 2*n* – 2 non-terminal nodes, while the minimum BDDs of symmetric bent functions were found to have 4*n* – 8 non-terminal nodes. Furthermore, the simple observation of the BDDs of the symmetric bent function demonstrates the cyclic nature of symmetric functions. Specifically, the functions are dependent entirely on the number of variables set to 1, and not the order in which the variables are listed.

The BDDs of the homogeneous functions of 6-variables, while not nearly as aesthetically pleasing as the DQFs and symmetric bent functions, were still quite revealing. All 30 functions were revealed to have identical BDD structures. That is, the total number of nodes per level in each BDD was identical. From this realization, it was discovered that P-equivalent functions will have identical BDDs for distinct variable orderings.

Finally, the BDDs of functions in the same affine class were shown to have identical structures. Functions in the same affine class are not P-equivalent, so the BDDs themselves were not identical, but this discovery demonstrated that all functions in an affine class have BDDs of the same size.

The research demonstrated that many bent functions, despite their nonlinearity, have predictable characteristics. However, only a small sample of bent functions was investigated. Further research may yield more comprehensive results. The creation of a program capable of finding the minimum BDD of any Boolean function may also prove broadly useful to others interested in BDD construction.

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

A. PROBLEM DEFINITION

Most modern cryptographic systems rely on Boolean functions as a part of the cipher process. In order to be effective, these Boolean functions must exhibit certain properties that aid in the obfuscation of the cryptographic key to an attacker. Amongst other properties, such as balancedness or low autocorrelation, ciphers benefit from Boolean functions with a high nonlinearity.

Boolean bent functions have the unique property of having the highest nonlinearity for any given function of *n*-variables [1]. Bent functions are also extremely rare and difficult to enumerate, making the discovery of unique bent functions for large *n* quite valuable. Despite their nonlinearity, bent functions alone do not make cryptographically sound Boolean functions, because they lack balancedness. However, bent functions make an excellent starting point for modification into a cryptographically sound Boolean function [2].

One method of representing a Boolean function is through a Reduced Ordered Binary Decision Diagram (ROBDD, or BDD for short). In a binary decision diagram, the function is represented graphically in a binary tree structure. Each level of the tree corresponds to a variable of the function, and each edge between vertices represents the decision of either a binary 0 or 1 for that variable. Thus, by setting each input variable to either a 0 or 1, the tree can be traversed, revealing the output of the function for that specific combination of input variables. BDDs can be useful because many human-designed functions have simple BDDs. That is, they tend to have a relatively small number of vertices, or nodes. Representing a function in this manner can result in speedy computation of functions, or reveal interesting characteristics when viewed graphically [3].

B. THESIS GOALS

This thesis analyzes a subset of well-known bent functions with the intention of identifying their characteristics with regard to their respective binary decision diagrams. This approach is taken because bent functions are inherently complex, and it is believed that research of this nature has never been attempted.

In particular, disjoint quadratic functions, homogeneous bent functions of algebraic degree three, and symmetric bent functions are analyzed in depth. Where applicable, patterns in these functions' structures are commented upon, and observations on the minimum size and characteristics of these functions are discussed.

An additional goal of this thesis is to develop a graphical program that displays an easily readable binary decision diagram of any given function. This program will be useful for future research projects on the topic of binary decision diagrams.

C. THESIS ORGANIZATION

Chapter I focuses on the general overview of the problem and presents the goal of the thesis. Chapter II provides background on Boolean functions, as well as defining bent functions and introducing the difficulties in enumerating those functions. Chapter III discusses binary decision diagrams in greater detail, indicating their current and future applications. Chapter IV describes the program used in this thesis to model the binary decision diagrams, and touches on algorithms that may be relevant to future researchers. Chapter V analyzes the binary decision diagrams of a subset of the known bent functions. Chapter VI summarizes the findings of Chapter V and discusses improvements that can be implemented on the program designed for this thesis as well as future research that can be attempted on binary decision diagrams of bent functions and functions with other cryptographic properties.

II. BOOLEAN FUNCTIONS AND THEIR CRYPTOGRAPHIC PROPERTIES

A. BOOLEAN FUNCTIONS

A Boolean function *f* of *n* variables is defined as

$$f : \mathbb{F}_2^n \to \mathbb{F}_2, \mathbb{F}_2 = \{0,1\},$$

where $\mathbb{F}_2^n$ represents the vectorspace of dimension *n* of the binary field $\mathbb{F}_2$ [2]. Boolean functions are commonly represented by a truth table, which represents the (0,1) value for each combination of *n* input variables, each also set to (0,1). In other words, the truth table is the (0,1) sequence defined by ($f(\mathbf{v_0})$, $f(\mathbf{v_1})$,…, $f(\mathbf{v_{2^\wedge(n-1)}})$) where $\mathbf{v_0}$ represents the *n* variables defined as (0,…0,0), $\mathbf{v_1}$ = (0,…0,1), and $\mathbf{v_{2^\wedge(n-1)}}$ = (1,…1,1), ordered lexicographically [2]. This paper will often refer to a truth table simply as TT.

	x_1	x_2	$x3$	***f***	
$\mathbf{v_0}$→	0	0	0	**0**	←$f(\mathbf{v_0})$
$\mathbf{v_1}$→	0	0	1	**1**	←$f(\mathbf{v_1})$
$\mathbf{v_2}$→	0	1	0	**1**	←$f(\mathbf{v_2})$
$\mathbf{v_3}$→	0	1	1	**0**	←$f(\mathbf{v_3})$
$\mathbf{v_4}$→	1	0	0	**1**	←$f(\mathbf{v_4})$
$\mathbf{v_5}$→	1	0	1	**0**	←$f(\mathbf{v_5})$
$\mathbf{v_6}$→	1	1	0	**0**	←$f(\mathbf{v_6})$
$\mathbf{v_7}$→	1	1	1	**1**	←$f(\mathbf{v_7})$

Table 1. Truth table for the three variable exclusive-or function. The column underneath *f* represents the truth table of the Boolean function.

1. Algebraic Normal Form

Boolean functions can also be represented in Algebraic Normal Form (ANF), a standardized method for representing Boolean functions. Functions represented in this form have the benefit of allowing one to identify its linear characteristics more readily than its truth table form. A Boolean function *f* of *n* variables can be represented in ANF as

$$\begin{aligned} f(x_1, x_2, ..., x_n) = a_0 \oplus & \\ & a_1x_1 \oplus a_2x_2 \oplus \cdots \oplus a_nx_n \oplus \\ & a_{1,2}x_1x_2 \oplus \cdots \oplus a_{n-1,n}x_{n-1}x_n \oplus \\ & \cdots \oplus \\ & a_{1,2,...,n}x_1x_2 \ldots x_n, \end{aligned}$$

where the coefficients $a_i \in \{0,1\}$. In the previous example of a three variable exclusive-or function, the coefficients a_1=a_2=a_3=1, and all other coefficients a_i = 0. Thus, the ANF of the exclusive-or function is

$$f(x_1, x_2, x_3) = 1 \cdot x_1 \oplus 1 \cdot x_2 \oplus 1 \cdot x_3 = x_1 \oplus x_2 \oplus x_3 .$$

2. Algebraic Degree

The algebraic degree of a Boolean function is the number of variables in the highest order monomial of the ANF with a nonzero coefficient [2]. In the function

$$f = x_1x_2 \oplus x_3x_4x_5$$

the algebraic degree is 3 due to the monomial $x_3x_4x_5$.

3. Linear Functions

A linear function is any function of algebraic degree one and for which a_0=0. The notation for a linear function is commonly given as

$$\ell_a(x) = a \bullet x = a_1x_1 \oplus a_2x_2 \oplus \cdots \oplus a_nx_n,$$

where each coefficient $a_k \in \{0,1\}$ for $1 \leq k \leq n$. Note that the constant function 0 is considered a linear function [2].

4. Affine Functions

Affine functions are all of the linear functions and their complements. Affine functions can be written as

$$\ell_{a,c}(x) = a \bullet x \oplus c,$$

where $c \in \{0,1\}$ [2]. Note that the constant functions 0 and 1 are both considered affine functions.

5. Hamming Weight

The Hamming weight, denoted by *wt*(*f*), of a function *f* is the number of nonzero values in its truth table. For a Boolean function, the Hamming weight is simply the number of ones in the truth table. In the case of the exclusive-or function of three variables, the Hamming weight is 4, since there are four ones in the function, as shown in Table 1.

6. Hamming Distance

The Hamming distance between two functions is the number of truth table elements that differ from each other. Between two functions *f* and *g*, the Hamming distance can be defined as

$$d(f,g) = wt(f \oplus g)$$ [2].

B. CIPHERS

In order to stand up to a variety of cryptanalysis attacks, the Boolean functions used to generate a cryptographic key must satisfy several, often-conflicting properties. Properties that may help reduce vulnerability to one attack may not be useful against another. Before going on, a few definitions:

1. Plaintext

Plaintext in a cryptography scheme represents the raw information or data that a user intends to transmit. Plaintext has not yet been altered by an encryption process.

2. Cryptographic Key

The cryptographic key is the parameter that modifies the plaintext to generate an encrypted ciphertext. Generally, keys should be large enough such that the key cannot be guessed by an attacker through enumeration, or brute force. An example of a cryptographic key is the 256-bit key assigned by a user of Wi-Fi Protected Access / Pre-shared Key (WPA2/PSK) in many wireless networks.

3. Ciphertext

Ciphertext is the output of a cryptographic process after a cryptographic key has been used to modify the original plaintext data.

4. Cipher Example

In order to demonstrate the application of a key to generate ciphertext, a simple stream cipher will be discussed.

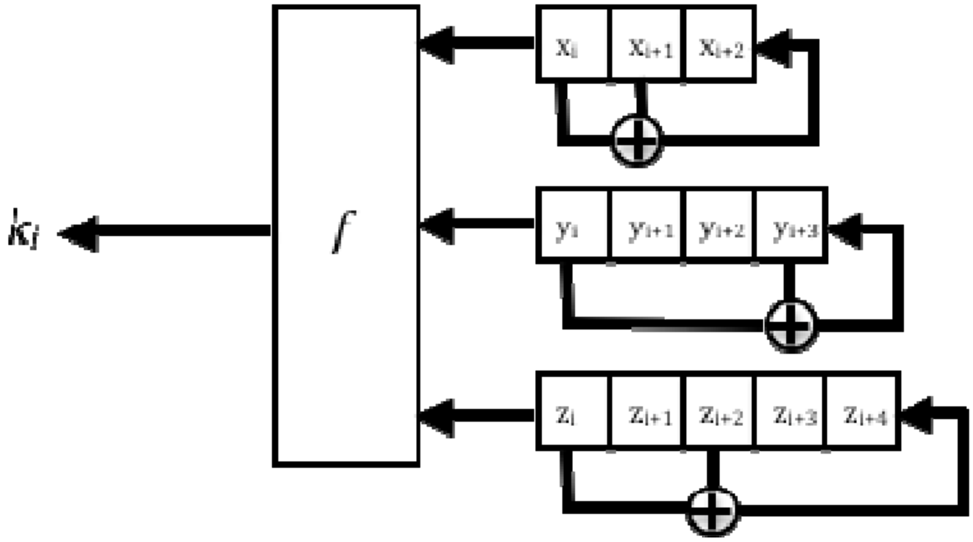


Figure 1. A simple cipher using a 12-bit key.

Since linear feedback shift registers (LFSRs) are easy and cheap to implement, they are frequently used in pseudo-random key generators. In Figure 1, a 12-bit key is spread amongst three "x" bits, four "y" bits, and five "z" bits. The XOR functions shown allow each sequence of bits to generate a pseudo-random pattern. The leading bits of each LFSR (x_i, y_i, and z_i) are then sent to *f*, which, in most cases, represents a Boolean function applied to each of the incoming stream of bits. This function *f* could be as simple as an XOR operation, but generally a more cryptographically secure Boolean function is chosen. The output, k_i, is then combined with a plaintext bit, typically via an XOR operation to generate one bit of ciphertext. The LFSRs compute the next series of bits in the keystream and the process is repeated until all plaintext has been encrypted. In order for the receiver to decrypt the ciphertext into a plaintext form, knowledge of the function *f*, the LFSR patterns, and the initial bits in the keystream must be known. These are the aspects of a cipher for which cryptographically secure properties are desirable.

5. Cryptographic Properties

When constructing a cryptographic system, the cipher's primary characteristics should be to create diffusion and confusion. *Diffusion* helps to mask the key and the plaintext data by dissipating the statistical properties of the ciphertext. In other words, each bit of the cipher function should affect as many bits of the plaintext as possible to spread the statistical significance of each bit in the ciphertext over a wide range. Ideally, for every bit change in the plaintext, a good cipher would change exactly half of the bits in the final cipher text, making the structure of the plaintext more difficult to detect [4, 5]. *Confusion*, as described by Claude Shannon, is "the method to make the relation between the simple statistics of the ciphertext and the simple description of the key a very complex and involved one." Ultimately, the goal of a confusing cipher is to make it difficult to identify the encryption key even if multiple ciphertext-plaintext pairs have been identified.

In general, properties identified as useful for cryptographic purposes are balancedness, low autocorrelation, correlation immunity, algebraic immunity, and nonlinearity. The majority of these properties will not be defined here, as they are not particularly relevant to the research conducted for this thesis, which will focus on the very desirable property of nonlinearity, specifically in regard to the study of bent functions. However, extensive studies have been done on the other cryptographic properties mentioned by other researchers, and the reader is directed to [2] and the references therein for further information.

It is worth mentioning that tradeoffs are required for any cryptographic function, as no Boolean function can exemplify all of the cryptographic properties. For example, bent functions, which have maximum nonlinearity, are never balanced. Balanced functions are those that contain an equal number of 1s and 0s in their truth tables, and help satisfy the cryptographic property of diffusion. Since bent functions can never be balanced [1, 2], a bent function is rarely used in an encryption device. It is possible, instead, to modify a bent function such that it becomes balanced, at some cost to its nonlinearity.

C. BENT FUNCTIONS

As previously mentioned, bent functions are those that have the highest possible nonlinearity for a function of *n*-variables. That is, bent functions have the highest possible Hamming distance from the affine functions of *n*-variables. This property of nonlinearity is important in generating confusion in a cryptographic process, since linear cryptanalysis attacks are capable of breaking most highly linear systems easily by using an affine function to approximate the actual function used [6]. For this reason, bent functions are useful in the generation of cryptographic ciphers.

1. Definitions

a. Nonlinearity

In a paper by Butler and Sasao [7], nonlinearity is succinctly defined as:

The nonlinearity NL_f of a function *f* is the minimum number of truth table entries that must be changed in order to convert *f* to an affine function.

Nonlinearity can also be defined as the minimum Hamming distance between the truth tables of *f* and an affine function.

$$NL_f = \min(wt(f \oplus a_1), wt(f \oplus a_2), ..., wt(f \oplus a_k)),$$

where a_i is an affine function indexed by $1 \le i \le k = 2^{n+1}$.

The 3-variable function $f = x_1x_2x_3$ has a nonlinearity of 1. The AND function *f* has a solitary 1 in its truth table, which has a Hamming distance of 1 from the linear function of constant 0, and of course, *f* is not itself affine [7].

b. Bent Functions

Let *f* be a Boolean function on *n*-variables, where *n* is even. *f* is a bent function if its nonlinearity is $2^{n-1} - 2^{\frac{n}{2}-1}$ [7].

Bent functions can also be defined by their Walsh transform coefficients. Specifically, a Boolean function *f* in *n* variables is called bent if and only if the Walsh transform coefficients of $\hat{f}$ are all $\pm 2^{n/2}$, that is, $W(\hat{f})^2$ is constant. $\hat{f}$ represents the sign function of function *f,* and *W*() is the Walsh transform [2]. This definition is useful when using bent functions in spread spectrum applications. This definition in terms of Walsh transform coefficients is only offered here for completeness; for the sake of this thesis, nonlinearity is the primary concern.

2. The Difficulty of Discovering Bent Functions

Despite bent functions' inherent usefulness in cryptographic applications, they are notoriously difficult to find. Although there are a few well-known classes of bent function that can be constructed for any number of *n*-variables, the majority of functions can only be found through computational enumeration. That is, the truth table of every function of *n*-variables must be sequentially checked against the known affine functions for nonlinearity. This process is time consuming, as there exist 2^{2^n} functions to compare! Even for the relatively small set of Boolean functions of 6-variables, there exist1.84×10^{19} functions to enumerate in search for the entire set of bent functions!

Complicating matters further, the number of bent functions of *n*-variables is unknown for general *n*. Upper and lower bounds have been identified, but the exact number of bent functions of *n*-variables is unknown.

a. Lower Bound

It has been shown by [8, 2] that rows in the Sylvester-Hadamard matrix yield a bent function when concatenated. The definition of the Sylvester-Hadamard matrix and the derivation of the findings are beyond the scope of this thesis. Nevertheless, it has been shown that, for $n = 2k$, the concatenation of the 2^k Sylvester-Hadamard rows or their complements in arbitrary order results in $(2^k)!2^{2^k}$ different bent functions of n-variables [8, 2]. Thus, there exist at least $(2^k)!2^{2^k}$ bent functions for $n = 2k$ variables.

b. Upper Bound

In [1] it was shown that the maximum algebraic degree of a bent function is $n/2$ for $n > 2$. This implies that the algebraic normal form has

$$\sum_{i=0}^{n/2} \binom{n}{i} = 2^{n-1} + \frac{1}{2}\binom{n}{n/2} \text{ coefficients,}$$

any of which can be either 0 or 1 [2], while all other coefficients must be 0. Thus, the number of functions that can be derived from these coefficients is

$$2^{2^{n-1}+\frac{1}{2}\binom{n}{n/2}},$$

which represents the upper bound of total possible bent functions of n-variables.

c. Enumerations

Due to the difficulties described in identifying the truth table or even the total number of bent functions, the total set of bent functions is only known up to 8-variables. Although bent functions are cryptographically desirable due to their nonlinearity, they have the added benefit of being rare and difficult to find. The lack of knowledge of higher order functions makes their employment particularly useful. Still, the following chart should illuminate the difficulties in discovering a bent function.

n	Lower bound	# of bent functions	Upper Bound	# of Boolean functions	Fraction of bent functions
2	8	8	8	16	2^{-1}
4	384	896	2048	65536	$2^{-6.2}$
6	$2^{23.3}$	$2^{32.3}$	2^{38} [9]	2^{64}	$2^{-31.7}$
8	$2^{95.6}$[10]	$2^{106.291}$[10]	$2^{129.2}$[10]	2^{256}	$2^{-149.7}$

Table 2. Known bent function quantities (After [2]).

Table 2 helps to illustrate the difficulties described above. For instance, there are $2^{256} = 1.158 \times 10^{77}$ Boolean functions of 8-variables. Assuming that a 2.4GHz computer could identify a function as bent or not at the rate of one function per clock cycle, it would still take 1.53×10^{60} years to search the entire solution-space. It is for this reason that the bent functions are relatively unknown for $n \geq 10$. Furthermore, it is significant to note that, as the number of variables increases, the proportion of bent functions as compared to the total number of Boolean functions decreases quite rapidly. This makes the short-term yield of sequential or random enumeration of Boolean functions in an effort to exhaustively search quite low.

3. Known Bent Functions

Even though bent functions are extraordinarily difficult to find within the total search space, there are many known classes that can be constructed for any order n-variables. A few of those classes will be discussed here.

a. Disjoint Quadratic Functions

Disjoint Quadratic Functions (DQFs) are functions of the form

$$f(x_1, y_1, ..., x_k, y_k) = \sum_{i=1}^{k} x_i y_i = x_1 y_1 \oplus x_2 y_2 \oplus \cdots \oplus x_k y_k$$

and were shown in Rothaus' original paper to be bent [1]. Rothaus called these functions members of FAMILY I, and are considered the simplest general form of bent functions

[11]. FAMILY I bent functions are also called the dot product, and can be written as

$$f(x,y) = x \bullet y.$$

FAMILY II bent functions are related in that they are a DQF concatenated with any function of half the variable set. That is, FAMILY II functions can be written as

$$f(x,y) = x \bullet y \oplus g(x) \text{ or } f(x,y) = x \bullet y \oplus g(y),$$

where *g* is any arbitrary function. Note that *g* must be composed specifically of the variables associated with *x* or the variables associated with *y* and cannot be mixed [1, 11].

b. Symmetric Bent Functions

A symmetric Boolean function is unchanged by any permutation of the input variables. An example of a symmetric function is the AND operation. Regardless of the ordering of the variables, the truth table will always be {0,0,0,1}. Due to dependence of symmetric functions on only the number of variables set to 1, symmetric functions can be represented in a condensed truth table. Tables 3 and 4 serve to illustrate this property. Note specifically how the number of variables column of Table 3 corresponds with the output value in both tables.

# of Variables set to 1	*f*
0	0
1	1
2	0
3	1
4	1

Table 3. Condensed Truth Table of an Arbitrary Four-Variable Symmetric Function

x_1	x_2	x_3	x_4	***f***
0	0	0	0	**0**
0	0	0	1	**1**
0	0	1	0	**1**
0	0	1	1	**0**
0	1	0	0	**1**
0	1	0	1	**0**
0	1	1	0	**0**
0	1	1	1	**1**
1	0	0	0	**1**
1	0	0	1	**0**
1	0	1	0	**0**
1	0	1	1	**1**
1	1	0	0	**0**
1	1	0	1	**1**
1	1	1	0	**1**
1	1	1	1	**1**

Table 4. Full Truth Table of the same Four-Variable Symmetric Function

Bent functions that are also symmetric are interesting in that there are only four possible functions [12], regardless of the number of variables. All symmetric bent functions have the form

$$f(x_1, x_2, \ldots, x_n) = \sum_{1 \le i < j \le n} x_i x_j \oplus c \sum_{i=1}^{n} x_i \oplus d,$$

where $c, d \in \{0,1\}$. The four different functions result from the four assignments of values to *c* and *d*.

In the simplest case, both *c* and *d* are set to 0, resulting in the symmetric bent function of the form

$$f(x_1, x_2, ..., x_n) = \sum_{1 \le i < j \le n} x_i x_j = x_1x_2 \oplus x_1x_3 \oplus \cdots \oplus x_1x_n \oplus x_2x_3 \oplus \cdots \oplus x_{n-1}x_n.$$

This form is the XOR operation on all possible pairs of variables. Since the XOR operation results in 1 only when there are an odd number of product terms of value 1, it has been proven previously that the condensed truth table of the function takes the form

$$c_k = \binom{k}{2} (\mathrm{mod}\, 2),$$

where c_k is the output of the condensed truth table for *k* variables set to 1 [2]. Table 5 illustrates the pattern the condensed truth table follows, namely as the variables set to 1 increases incrementally, the condensed truth table follows the repeating pattern {0, 0, 1, 1}. Thus, a truth table can be constructed for a symmetric bent function of any size of *n*-variables.

# of Variables set to 1	*f*
0	0
1	0
2	1
3	1
4	0
5	0
6	1
7	1
…	…

Table 5. Condensed Truth Table of the Basic Form of a Symmetric Bent Function

c. Constructing Bent Functions

In addition to general forms of bent functions that can be constructed for functions of any size, some methods exist for constructing new bent functions from known bent functions.

(1) Concatenation. Given a function $f(x)$ and $g(y)$ that are both bent, the concatenation $h(x,y) = f(x) \oplus g(y)$ is also bent [2]. Note that x and y are disjoint sets of variables. Thus, if $f(x)$ and $g(y)$ are each 4-variable functions, the resulting function $h(x,y)$ is an 8-variable function.

(2) Affine Classes. Given a bent function f, the XOR of f and any affine function on the same set of variables is also bent. That is, $g(x) = f(x) \oplus \ell(x)$ is bent if f is bent and ℓ is affine [2]. It should be noted that $g(x)$ is considered to be in the same affine class as $f(x)$.

D. SUMMARY

In this chapter, Boolean bent functions were defined. Bent functions are valuable in cryptographic systems due to the protection they offer against linear cryptanalysis attacks. However, bent functions are rare and difficult to find. Since the binary decision diagrams (BDDs) of bent functions have never been specifically investigated, the next chapter will focus on the construction, characteristics, and applications of BDDs.

THIS PAGE INTENTIONALLY LEFT BLANK

III. BINARY DECISION DIAGRAMS

There are many methods for representing a Boolean function. The most common are truth tables, as discussed in Chapter II, and circuits. Circuit representation uses transistor gates to model the function in digital form. Truth tables are typically used in the design phase to identify characteristics of the function, and the circuit representation is used in the implementation of the function, such as in an integrated circuit or Field Programmable Gate Array (FPGA).

Another practical method of representing a Boolean function is by a Binary Decision Diagram (BDD), which organizes the function into a tree structure that can be traversed one variable at a time. Representing functions in this form is currently popular in many Computer Aided Design (CAD) applications [15]. The remainder of this chapter will describe the construction of BDDs and their valuable properties.

A. BINARY DECISION DIAGRAM CONSTRUCTION

In a BDD, a Boolean function is represented as a rooted, directed, acyclic graph in which each vertex, or node, has two edges, or links. Each node represents a variable (i.e., x_1, x_2, etc) and each edge represents a decision for that variable. Specifically, the edge represents either a 0 or 1 decision in the tree. Thus, the 0 edge for a node representing x_2 symbolizes the sub-function for which input variable x_2 is set to 0.

Typically, operations are performed on the BDD to reduce its complexity while still maintaining its canonical representation of the originating Boolean function. For clarity, this thesis will first describe BDDs for which no reductions are performed.

1. Binary Decision Tree

Although technically not considered a BDD, decision trees represent the simplest method of converting a Boolean function to a tree structure. A decision tree is a full, complete binary tree, meaning that each level of the tree is filled, and each node has two children. In other words, every binary decision tree of *n*-variables will have $2^n - 1$ non-terminal nodes, and 2^n terminal nodes. Non-terminal nodes are non-leaf nodes, and represent an input variable of the function. Terminal nodes are leaf nodes, and represent the output value of function for a specific combination of the input variables. Leaf nodes are those that have no children.

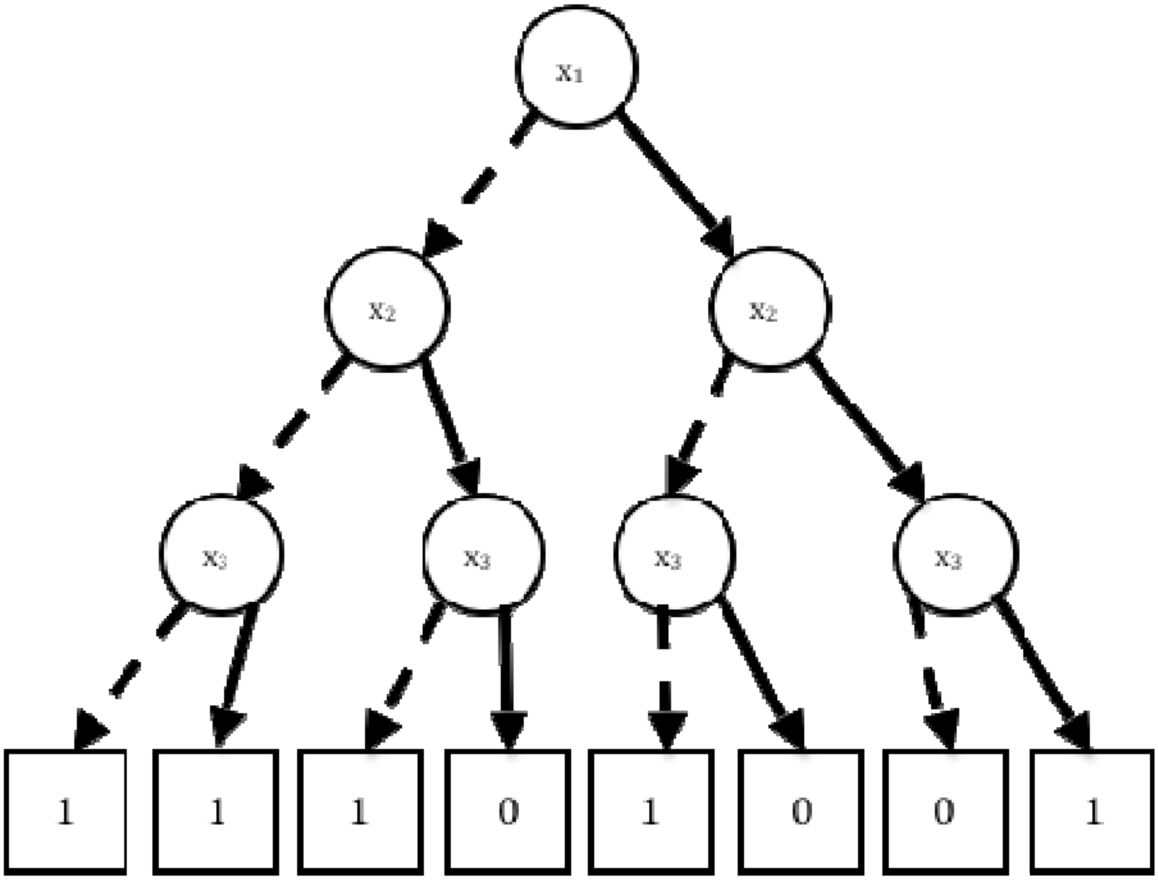


Figure 2. Binary Decision Tree of a 3-Variable Function

Figure 2 provides an example of a Binary Decision tree for 3-variables. Note that the final level of nodes simply duplicates the original truth table. The tree is traversed by starting at the root node, here represented by x_1, and choosing a path based on each variable's value. A variable value of 0 indicates that the dashed edge should be traversed, and a variable value of 1 indicates that the solid edge should be traversed. If the input variables are $x_1x_2x_3$ = 010, the tree is navigated by following the left edge from x_1, the right edge from the corresponding x_2 node, and left edge from the corresponding x_3 node. This results in a terminal value of "1" which corresponds with the output value for that specific input of variables, just as in a truth table.

2. Quasi Reduced Ordered Binary Decision Diagrams

A binary decision tree can be modified into a Quasi Reduced Ordered Binary Decision Diagram (QROBDD) by following a few simple rules.

a. The Ordered Property

An ordered BDD is one in which the variables in the tree are traversed in the same order regardless of the path chosen. That is, each level of tree can be said to represent one variable. Figure 2 is ordered in that the first level of the tree corresponds to x_1, the second level of the tree corresponds to x_2, and the third level corresponds to x_3. If the left sub-graph of Figure 2 is left untouched, but the right sub-graph modified such that there is an x_3 node in the second level and two x_2 nodes in the third level, the tree is no longer ordered. That is, traversing the left sub-graph would result in a different variable ordering than the right sub-graph [13].

The ordered property does not imply that the variables must be ordered numerically. A tree that traverses variables in the sequence $x_3x_1x_2$ is still considered ordered provided that all paths result in the same variable ordering.

b. The Quasi Reduced Property

A binary decision diagram is considered quasi reduced if its isomorphic sub-graphs are merged such that there are no redundancies. Isomorphic sub-graphs are those that are considered to be equivalent. That is, the portions of the graph that yield the same sub-function are isomorphic. This reduction is the simplest compression that can be applied to a BDD [14].

The binary decision tree in Figure 2 will now be quasi reduced to demonstrate this property. The first easily identifiable isomorphic sub-graphs are the terminal nodes: 0 and 1. Each individual 0 and 1 need not be repeated; one of each will suffice. Figure 3 reduces these sub-graphs:

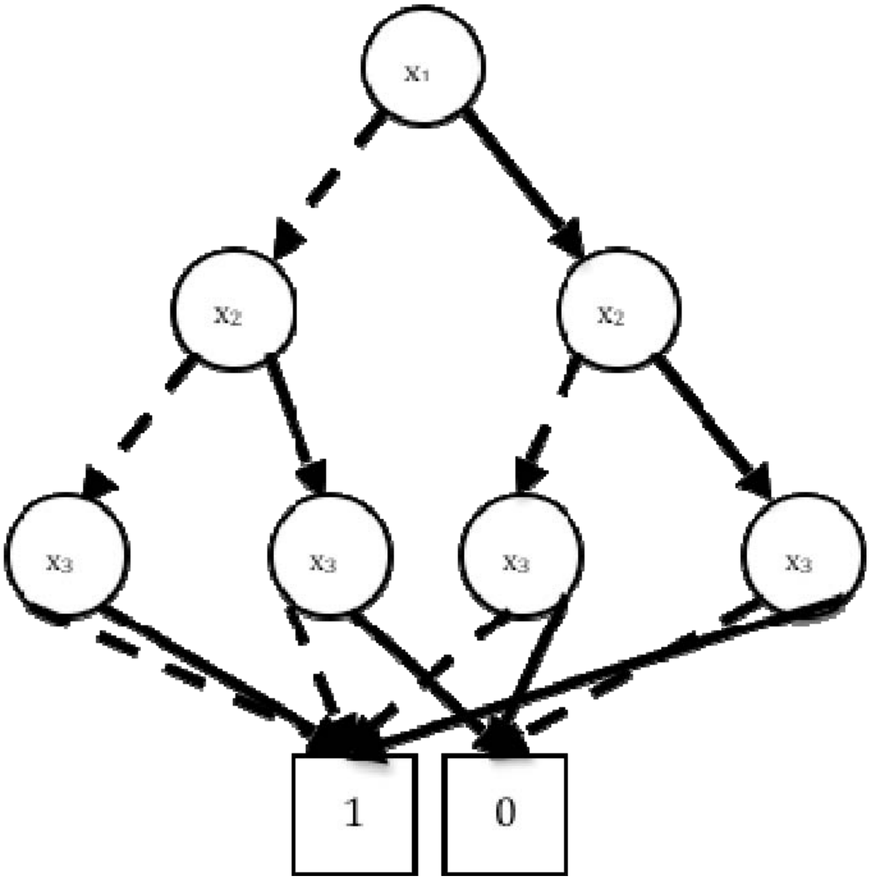


Figure 3. BDD with reduced terminal nodes.

Reducing the terminal node sub-graphs had a significant impact on the size of the BDD, removing six nodes. Further inspection of Figure 3 reveals that amongst the level 3 sub-graphs (variable x_3) the two nodes in the middle are isomorphic. Note that both nodes result in a 1 output when x_3 is set to 0, and a 0 output when x_3 is set to 1. Neither of the other nodes shares this specific property. Thus, the two middle x_3 nodes are isomorphic with one another. Figure 4 will demonstrate the reduction:

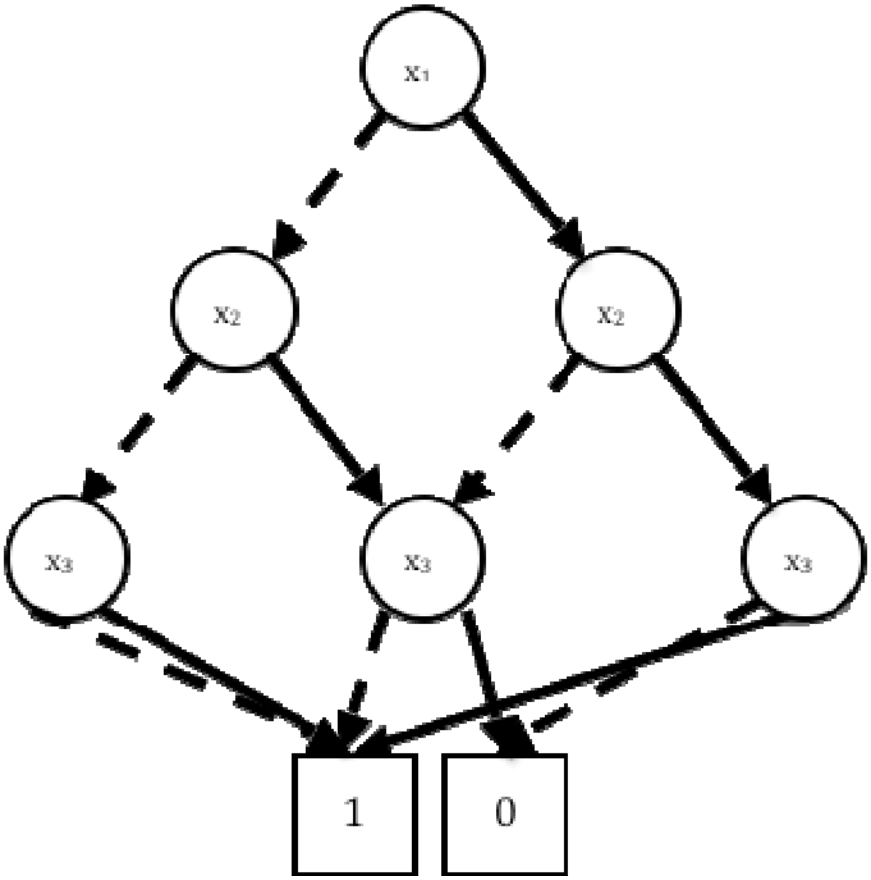


Figure 4. BDD with all isomorphic sub-graphs merged.

Merging of the x_3 isomorphic sub-graphs resulted in another node reduction. Note that the BDD in Figure 4 now has only eight nodes, whereas the original tree contained fifteen. Careful inspection of Figure 4 reveals that no more isomorphic sub-graphs remain. Each node is unique. Thus, Figure 4 represents a quasi-reduced ordered BDD.

3. Reduced Ordered Binary Decision Diagrams

Reduced Ordered BDDs (ROBDDs) remove further redundancies from a decision tree, but are somewhat more complex in that they can remove nodes that are isomorphic sub-graphs of themselves. That is, if both edges from a single node are directed to the same location, that node is redundant. The node in question is then removed, and its parent node is redirected to point to the removed node’s child. The difficulty in this method is that a child node can be located several levels below its parent. This requires each node to have knowledge of its own level in the tree [13, 14]. Under the QROBDD rules, each variable would be traversed in every path to the terminal nodes. ROBDDs, on the other hand, are likely to skip variables on some paths from the root to the terminal nodes.

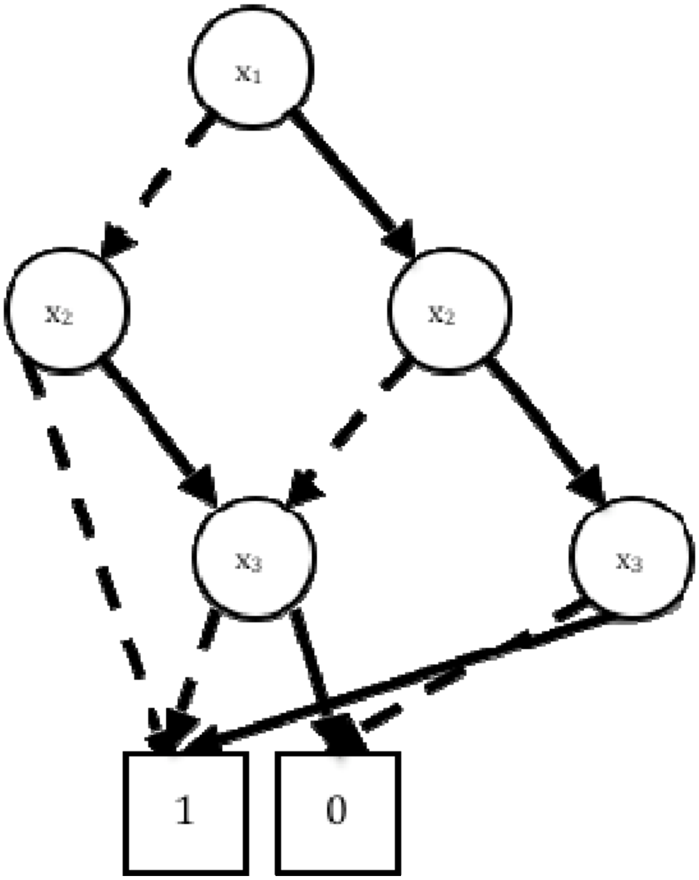


Figure 5. An ROBDD

Figure 5 modifies Figure 4 by removing the leftmost x_3 node. This was due to both edges of that node pointing to terminal node "1." The edge pointing to the removed node was then redirected to terminal node "1" instead. This has resulted in a completely irreducible ROBDD. Note that the x_3 level is skipped in the event that $x_1x_2 = 00$. It is also significant to note that the ROBDD has a total of 7 terminal and non-terminal nodes, which is a savings of more than 50% as compared to the full binary decision tree representation of Figure 2.

B. BINARY DECISION DIAGRAM COMPLEXITY

A Boolean function represented as a BDD has a relatively obvious complexity. Since most useful functions can be assumed to have both “0” and “1” terminal nodes, the complexity can be taken to simply be the number of non-terminal decision nodes in the graph.

Truth tables tend to make poor practical representation of a function since all truth tables have size exponential in *n*, the number of variables. That is, a function represented by a truth table must have 2^n elements, regardless of function. As an example, a 4-variable XOR function requires $2^4 = 16$ elements, like all 4-variable functions.

Boolean circuits, used in most digital applications, are statistically likely to be exponential as well [14]. In Boolean circuits, the number of gates necessary to fully realize the function represents the complexity. Gains are made in complexity compared to a truth table representation in that the switching nature of the logic gates tends to allow multiple outputs of the function to be determined by a single gate. Continuing the example of the 4-variable XOR function, it is well known that 4-NAND gates can represent a 2-variable XOR function. Three cascaded 2-variable XOR gates can make up a 4-variable XOR function. Thus, 12 NAND gates are required to represent a 4-variable XOR circuit. This is a 1/4 reduction in complexity as compared to a truth table representation.

Binary decision diagrams, just like circuits, also statistically tend to have exponential complexity. Gains can still be made compared to the truth table or gate representation. In particular, human designed Boolean functions tend to result in relatively simple BDDs, likely due to the useful properties of such simple functions [15]. By way of a final example, a 4-variable XOR function represented as a BDD requires only seven non-terminal nodes, a significant reduction as compared to a truth table or a circuit (Figure 6). It should be noted that a BDD is not guaranteed to have lower complexity than its equivalent Boolean circuit.

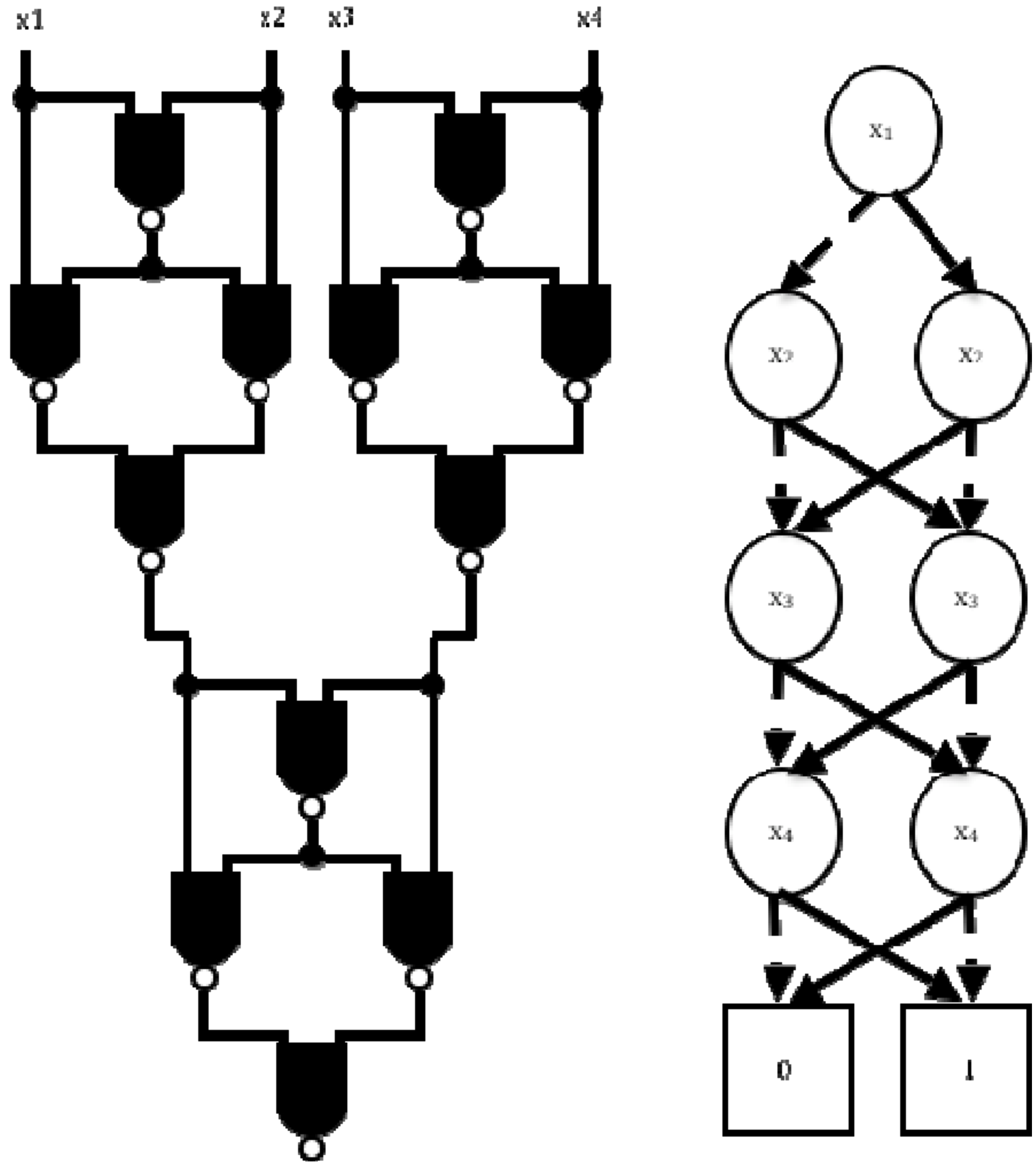


Figure 6. Side-by-side comparison of 4-variable XOR function as represented by Boolean circuit logic and a Binary Decision Diagram

1. Maximum Complexity of a Reduced Ordered Binary Decision Diagram

Although the proof is beyond the scope of this thesis, Michon, Yunes, and Valarcher identify the maximal complexity of a ROBDD of *n*-variables as

$$C_{ROBDD}(n) = 2^h + 2^{n-h} - 1,$$

where *h* is the height of inflexion and is a unique integer ≤ *n* such that:

$$2^{h-1} < 2^{n-(h-1)} \text{ and } 2^h \geq 2^{2^{n-h}} \text{ [14].}$$

Since the complexity is represented as a nested formula, the maximum complexities of the BDDs for up to ten variables are provided:

n	*h*	C_{ROBDD}
2	2	5
3	2	7
4	3	11
5	4	19
6	4	31
7	5	47
8	6	79
9	7	143
10	8	271

Table 6. Maximum complexity (number of nodes) in a BDD representative of a Boolean function of *n*-variables.

Although Table 6 describes the maximal complexity of any Boolean function, this thesis also concerns itself with symmetric Boolean functions. It has been shown that the number of nodes in the BDD of symmetric functions is asymptotic to $\frac{n^2}{2}$, as *n* increases. Furthermore, it has been shown that the average number of nodes in the BDD of a symmetric function is also $\frac{n^2}{2}$ [16]. The implication of this finding is that symmetric functions tend to have worst-case complexity within the bounds of the symmetric subset.

2. Variable Ordering

Unlike the truth table and circuit representations of Boolean functions, the variable ordering of a BDD can have a significant impact on the size of the graph. A good example of this is the case of the disjoint quadratic function mentioned in Chapter II. When the variables are ordered numerically, i.e., x_1, x_2, x_3, …, x_n, in the BDD, a minimum realization of the function is constructed. Note that in Figure 7 red edges indicate "0" decisions and blue edges represent "1" decisions. This minimum realization of a DQF requires 14 non-terminal nodes. It can be shown that for all DQFs of *n*-variables, the minimal BDD will have 2*n*-2 nodes [17].

Figure 8 reveals that when the variable ordering is altered to reflect a traversal order of $x_7 - x_1 - x_4 - x_6 - x_8 - x_2 - x_3 - x_5$, the size of the BDD balloons to 37 non-terminal nodes. In this instance, the variable ordering is considered to be *diabolical*, or the worst possible case. It is worth noting that not all functions have a diabolical variable ordering. For instance, the BDD of the 4-variable XOR function mentioned previously will always have the same structure regardless of variable ordering.

Since variable ordering is so significant in finding a minimal solution, it follows that testing all variable orderings would be desirable when constructing a BDD. Testing all variable orderings can quickly become impractical, since *n*! variable permutations exist. At *n*=10 – the maximum function size considered by this thesis - 3,628,800 permutations must be tested to ensure a minimum BDD realization. For this reason, heuristic methods such as linear sifting are necessary for minimum approximation of large order functions [18].

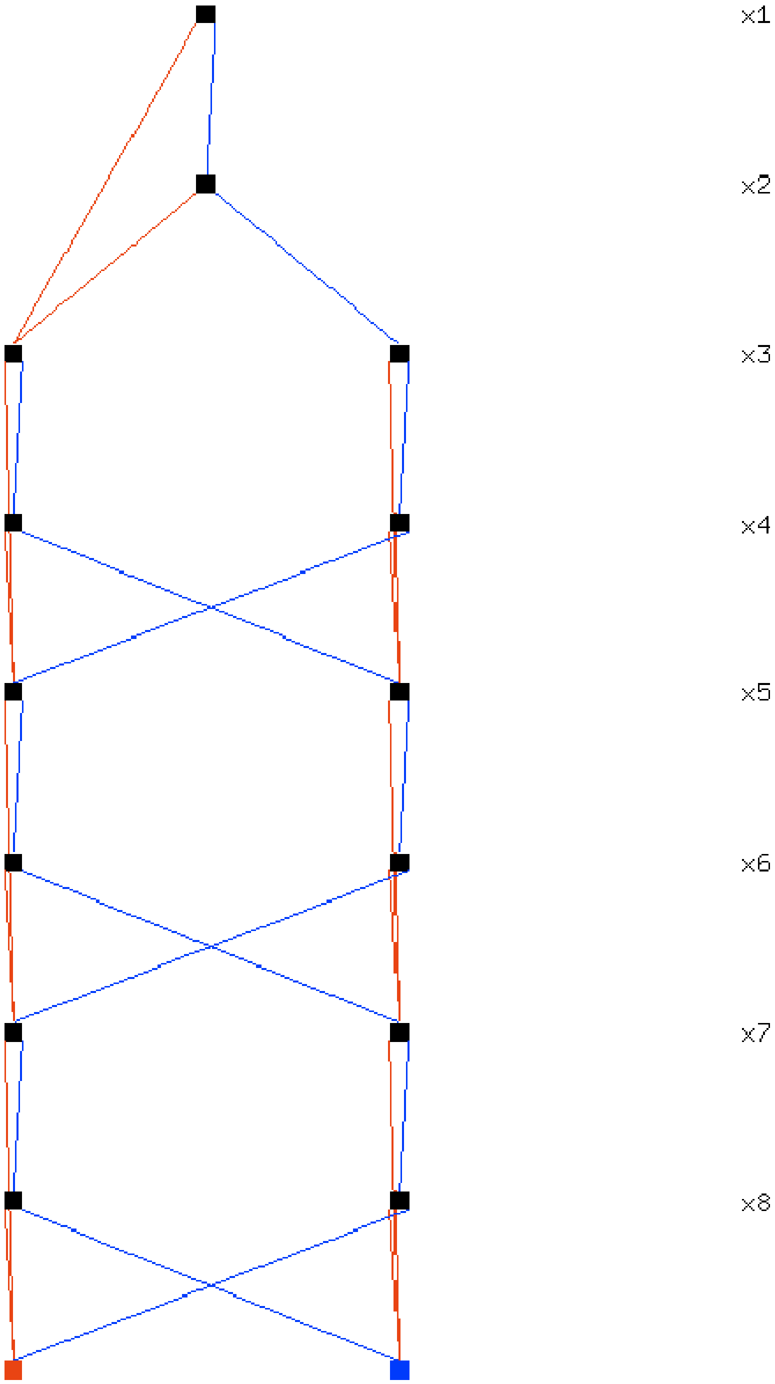


Figure 7. Minimum realization of DQF $x_1x_2 \oplus x_3x_4 \oplus x_5x_6 \oplus x_7x_8$.

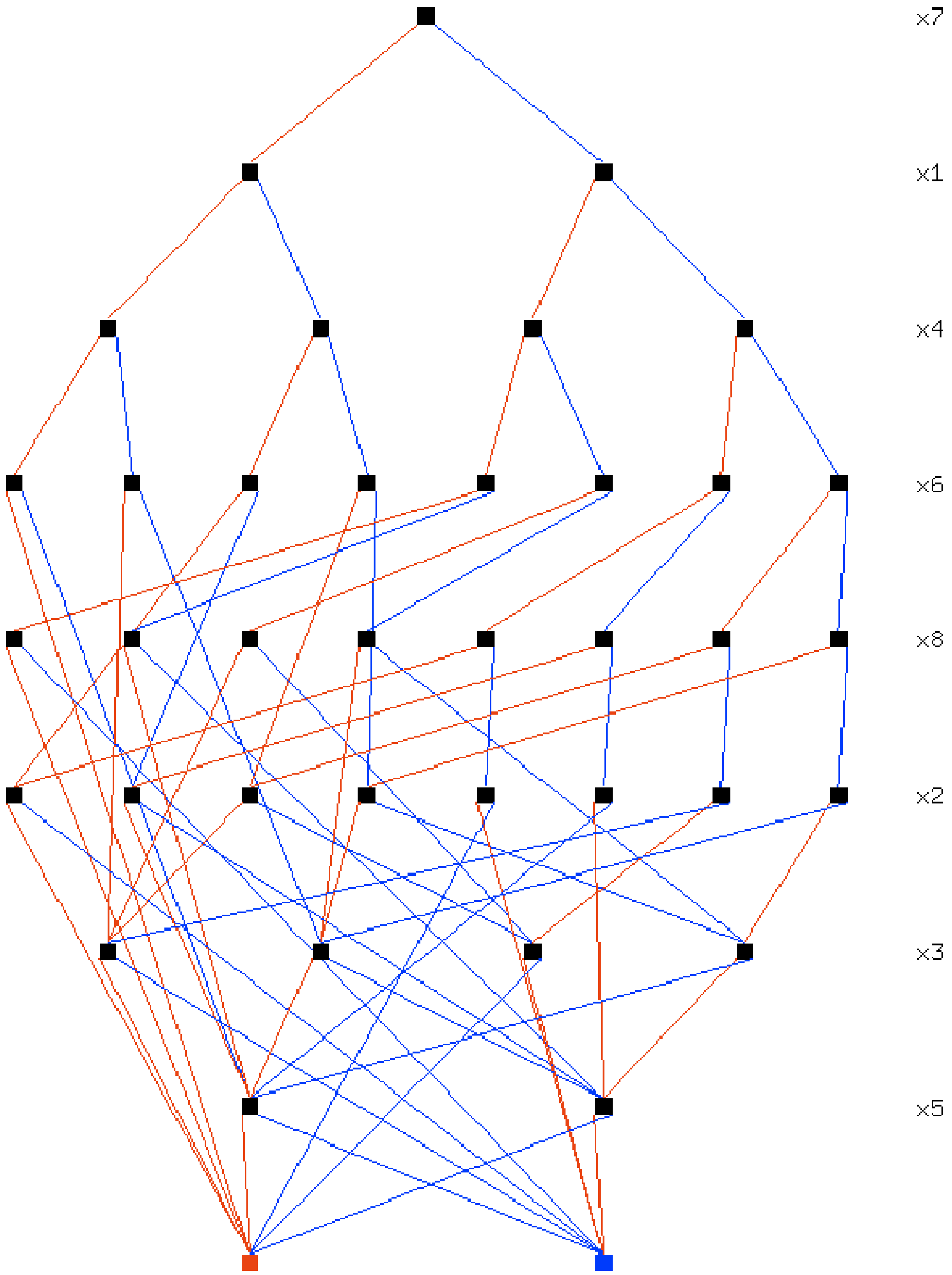


Figure 8. Maximum realization of DQF $x_1x_2 \oplus x_3x_4 \oplus x_5x_6 \oplus x_7x_8$.

C. BINARY DECISION DIAGRAM APPLICATIONS

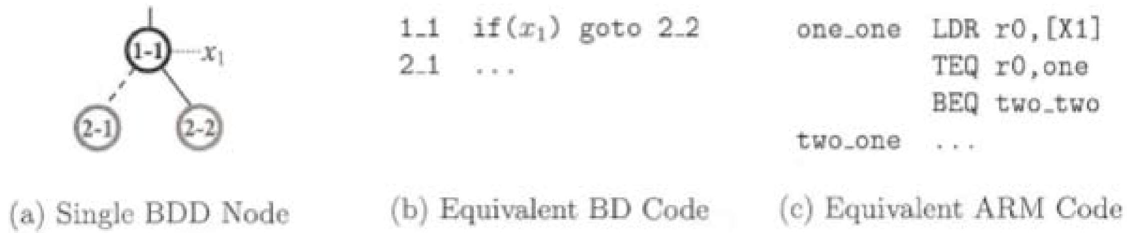
Due to the typically compact size of BDDs and their characteristic of easily representing sub-functions of a given Boolean function, Binary Decision Diagrams are popular in the use of VLSI (Very Large Scale Integrated Circuit) CAD (Computer Aided Design) applications [3, 15]. Furthermore, BDD nodes are easily implemented in conventional hardware, whether as a 2-to-1 multiplexer in traditional circuit design, or as a single 4-LUT (Look-up Table) in an FPGA (Field Programmable Gate Array) implementation.

Despite exponential complexity in the worst-case, BDDs are desirable in high-speed applications due to their linear delay. That is, BDDs of *n*-variables have a maximum delay of *n.* As an added benefit, as has been discussed in Section B, human designed functions tend to have lower than average complexity.

1. Binary Decision Machines

Due to the speed of traversing a BDD, they are useful in control and real-time applications [3]. General-purpose microprocessors are a bit clunky with regard to these applications due to the nature of their machine language. For instance, the "if-then-else" scenario of a BDD node would require three lines of machine code: A load register instruction, test instruction, and branch instruction [15].

Binary decision machines, specifically constructed for the purpose of interpreting binary decision diagrams, are capable of interpreting an entire "if-then-else" case in one instruction [3]. The simplified instruction set of such a machine is demonstrated in Figure 9.



(a) Single BDD Node

```
1_1  if(x1) goto 2_2
2_1  ...
```

(b) Equivalent BD Code

```
one_one  LDR r0,[X1]
         TEQ r0,one
         BEQ two_two
two_one  ...
```

(c) Equivalent ARM Code

Figure 9. Comparison of BD Machine Code and Microprocessor Machine Code [15]

It can also be shown that given a specific BDD, the minimum size of a BD Machine program is bounded. Specifically, the size of the program written in BD Code must have size $N \leq PSIZE_f \leq \frac{3}{2}N$ where N is the total number of BDD nodes [15]. The bounds of programs written with BDDs are significant, since a microcontroller application may have only a few kilobytes of storage space available. This is one reason why finding a minimum BDD is desirable.

2. Quaternary Decision Diagram Machine

A natural extension of the Binary Decision Machine is the Quaternary Decision Diagram (QDD) Machine. A QDD is identical to a BDD, except that each node can have up to four children. QDDs are desirable since they are capable of storing more information in each node, and therefore require a smaller number of nodes than a BDD. Since fewer nodes are traversed, a QDD optimized machine can evaluate a function faster than a BDD machine. However, the storage requirements of each node increase exponentially with the number of decisions available to it, and it is shown in [3] that QDDs have the minimum storage footprint.

In the QDD machine being developed by Renesas Technology Corporation, BDDs are still an essential element in the QDD design process. Specifically, the Boolean function in question is first characterized by a minimized BDD, and then variables are concatenated in order to construct the QDD [3]. Figure 10 illustrates this concept.

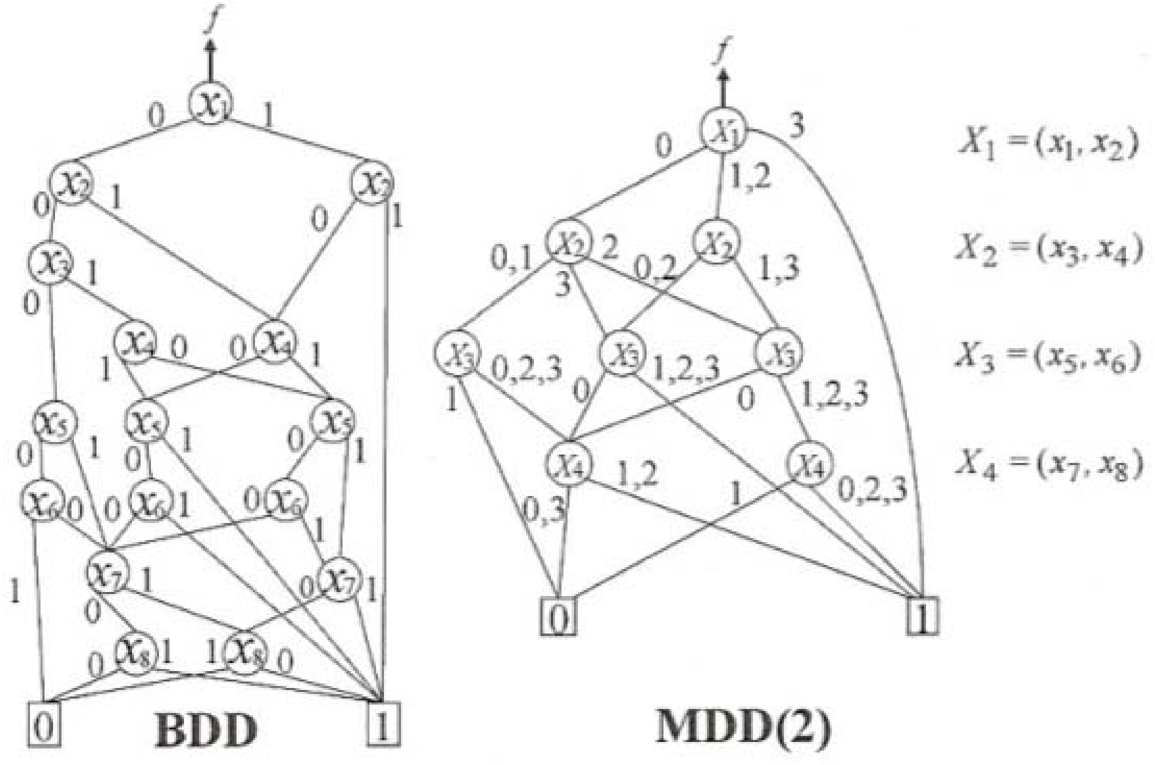


Figure 10. Creation of a QDD (labeled as MDD) from a minimized BDD [From 3].

Although QDDs tend to have roughly half the number of nodes as their equivalent BDDs, this does not necessarily mean a QDD implementation will be twice as fast. For instance, a BDD node can be processed more rapidly than a QDD node, due to its simplicity. Despite this, the QD machine being developed by Renesas Technology has typically performed 1.28 – 2.02 times faster than an equivalent BD code implementation [3].

D. SUMMARY

In this chapter, BDDs were introduced. Specifically, the method of construction was demonstrated, as well as their complexity. A brief discussion on current applications and research into BDD machines was also touched upon. The next chapter will describe the specific program designed for this thesis to rapidly generate BDDs.

THIS PAGE INTENTIONALLY LEFT BLANK

IV. BDDVIEWER

Several computer programs exist capable of converting a Boolean function into its binary decision diagram form. The Colorado University Decision Diagram (CUDD) Program [19] is one example of a popular and powerful program capable of manipulating a variety of decision diagram types. It is also capable of outputting each diagram in a format that can be interpreted as a visual graph by the program “dot” now incorporated into the GraphViz package [20]. Furthermore, CUDD can output a BDD in blif format, which presents the binary decision diagram as a series of 2-to-1 multiplexers, each corresponding to a node in the BDD.

Despite the availability of tools such as CUDD, it was decided that a new, self-contained visualization tool should be developed that directly addresses the questions at hand in this thesis. For instance, a graphically consistent visualization with intuitive edges and variable labels was highly desired. Furthermore, it was desired that during the BDD minimization process, the exact permutation of the variables would be reflected in the visualization, allowing the user to recognize which variables had been swapped to result in the resulting graph.

As a byproduct of starting a new program from scratch, the problem of representing a binary decision diagram in a conventional computer system imposed several challenges that resulted in some unique solutions. The specific algorithms used are discussed in the remainder of this chapter.

The program—known as BDDViewer—was written for the MacOS X operating system using Objective-C++ under Xcode v3.1.3. Graphics were provided using OpenGL due to the widespread support of those libraries. At the time of this writing, BDDViewer is not functional on a Windows operating system, although a port is in progress. The program was run on a 2.6 GHz Intel Core 2 Duo processor with 4 GB of RAM. It is believed that any modern 32-bit processor should run BDDViewer adequately, as the footprint of each BDD in RAM is on the order of kilobytes, and the graphics processes are trivial. It should be noted, however, that in order to find the minimum BDD realization of an n-variable function, the program must shift through every possible variable permutation, resulting in the construction of $n!$ BDDs. This can be quite computationally expensive, resulting in long waiting times for results of functions of large n, regardless of the CPU capabilities. For instance, running BDDViewer on the specifications provided resulted in a run time of two-and-a-half days to process all permutations of a 14-variable function. Due to the factorial nature of the algorithm, a 15-variable function would take 15 times as long, or over a month! Even computers running 2, 4, or even 8 times faster will hit a point in which finding all permutations is no longer practical shortly after the mid-teens.

A. REDUCTION PROCESS

The construction of a BDD is generally performed by evaluating a function as a series of sub-functions. The tree formed by the root node and all child nodes represents the function in its entirety. Each decision corresponds to a unique sub-function of the original. Assuming the variables are ordered numerically, traversing the “0” decision of the root (x_1) node results in a sub-tree that represents a sub-function for which x_1 is set to 0. Traversing the “1” edge results in a sub-tree that represents the sub-function for which x_1 = 0 and x_2 = 1. In this way, BDD construction can become quite tedious; since it requires a Boolean function to be evaluated $2^n - 1$ times before isomorphic sub-graphs can be identified and merged. This also causes the speed of the construction of a BDD to be dependent on the complexity of a Boolean function. For instance, an 8-variable OR function will be much less computationally expensive than an 8-variable majority function on a general processor.

1. A High Level View of the Problem

Inspection of the Boolean function’s truth table reveals that the shape of its BDD is fully realized by the string of output values. Table 7 represents a sample truth table for an arbitrary 4-variable function. Notice that the upper half of the truth table represents the sub-function for which x_1 = 0 and the lower half of the truth table represents x_1 = 1. Dividing each half of the truth table further splits the function into smaller sub-functions. This process is continued until the truth table is fully parsed.

x_1	x_2	x_3	x_4	***f***
0	0	0	0	**0**
0	0	0	1	**1**
0	0	1	0	**1**
0	0	1	1	**0**
0	1	0	0	**1**
0	1	0	1	**0**
0	1	1	0	**0**
0	1	1	1	**0**
1	0	0	0	**0**
1	0	0	1	**1**
1	0	1	0	**1**
1	0	1	1	**1**
1	1	0	0	**1**
1	1	0	1	**0**
1	1	1	0	**1**
1	1	1	1	**0**

Table 7. Arbitrary 4-variable function split into sub-functions

Each parsing of the truth table can be seen as a node in a tree. In particular, Table 7 represents a complete, full, binary decision tree as defined in Chapter III. Column x_1 represents the root node, with the thick double line splitting the upper and lower half into its two edges. Since the root node represents the entire function, this node can be labeled symbolically as the entire truth table output string “0110100001111010.” Likewise, column x_2 represents the two level 2 nodes. Each can be represented symbolically as the parsed truth table output “01101000” for the “0” edge and “01111010” for the “1” edge. Figure 11 demonstrates this explicitly with the first few levels of the binary decision tree.

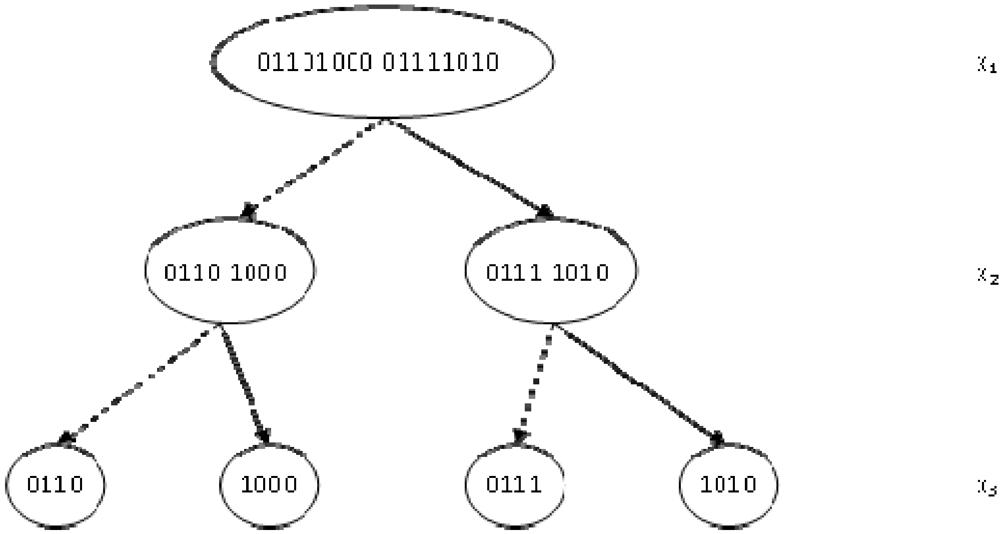


Figure 11. Partial binary decision tree constructed by parsing a truth table.

Viewing each truth table as representative of a sub-function allows one to make rapid observations about the nature of each sub-graph. For instance, the lower right node of Figure 11, labeled as “1010” and representative of the path $x_1 = x_2 = 1$, will be further parsed into two more nodes at level four of the graph, each of which represents a sub-function truth table of “10.” Since both of these nodes have identical truth tables, it can be concluded that they are isomorphic sub-graphs. Furthermore, node “1010” is an isomorphic sub-graph of itself, and is therefore redundant. Hence, the children are merged into one node, and node “1010” is removed from the tree entirely.

In short, each node can be derived from its parent node. The “0” decision represents the first half of the parent’s truth table, and the “1” decision represents the second half of the parent’s truth table. Any nodes that have identical truth tables will be merged. Sibling nodes with identical truth tables will be merged and the parent will be skipped. In this way, parsing a function’s truth table allows one to quickly and accurately generate a ROBDD without having to evaluate a function multiple times. In a high-level language, such as C++, this can easily be accomplished by representing a function’s truth table as a string type. Since strings have no limit on length and can be easily split into substrings, the size of a function represented in this manner is limited only by available memory size. The ideas presented can also be implemented at a lower-level, such as assembly language—with manipulations performed on bits, nibbles, bytes, half-words, and words— though the algorithms may be significantly more complex.

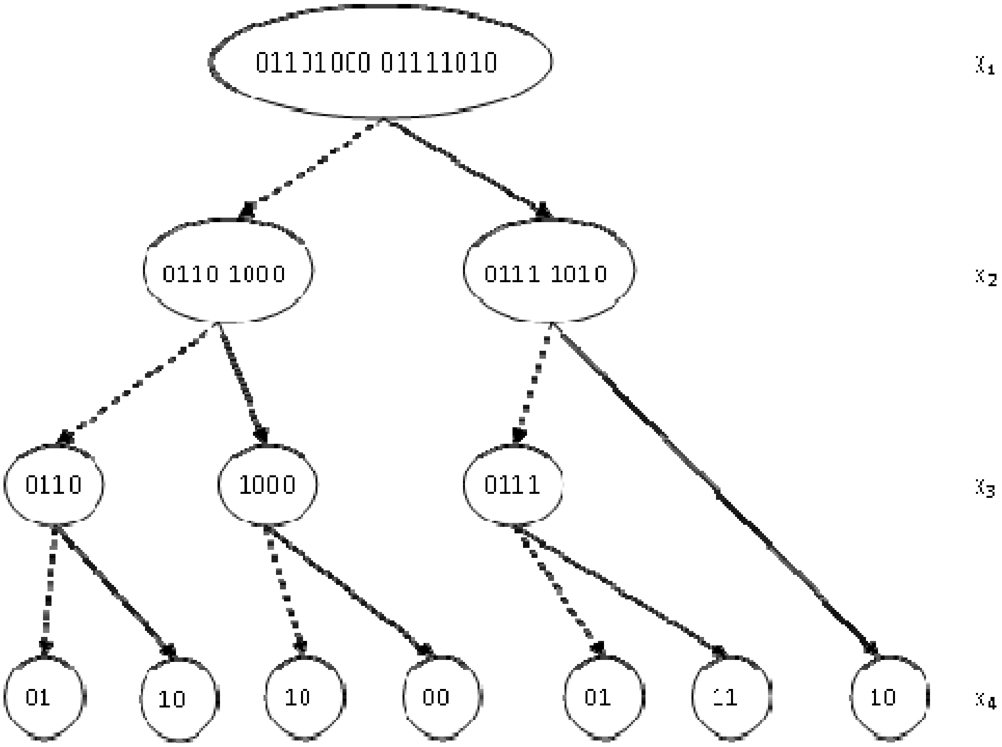


Figure 12. Partial BDD showing reduction of redundant node at level 3.

Figures 12 and 13 complete the algorithm. Note that in Figure 12, nodes represented by the sub-function “01” occur twice, and nodes represented by “10” occur three times. Since these are isomorphic, they will be merged into one node. The “00” and “11” nodes are clearly isomorphic of themselves, and will also be skipped. The final results of the process are shown in Figure 13.

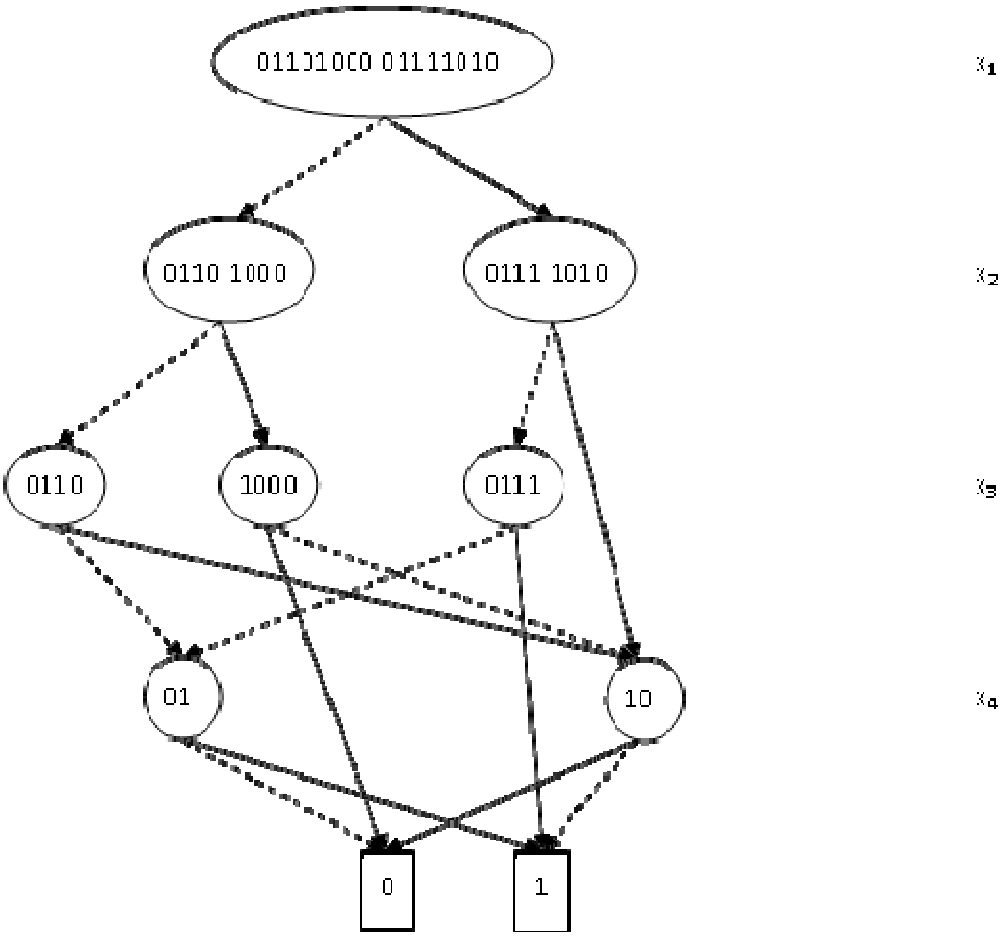


Figure 13. ROBDD generated by parsing truth tables and merging identical sub-functions.

2. Pseudo Code

```
createBDD(TT)

        thisTT = TT

        node = thisTT

        node.level = $log_2$(TT.size) – $log_2$(thisTT.size)

        if thisTT.size > 1 && left_side(thisTT) == right_side(thisTT)

                node = createBDD(left_side(thisTT))
```

```
else if thisTT.size > 1
    if left_side(thisTT) previously added
        left_child = address of previously added node
    else
         left_child =createBDD( left_side(thisTT))
    if right_side(thisTT) previously added
        right_child = address of previously added node
    else
        right_child = createBDD(right_side(thisTT))
```

B. MINIMIZATION

As mentioned in Chapter III, the size of a BDD is greatly affected by its variable ordering. Since minimized BDDs are of the greatest practical value in computing applications, a suitable minimization technique was required of the program. It was shown in Chapter III that an *n*-variable function has *n*! variable permutations, which hampers the ability to enumerate all orderings for large *n*. For instance, to find the minimum sized BDD of a 10-variable function would require the construction of 3,628,800 trees.

Fortunately for the sake of this thesis, bent functions are known for only relatively small values of *n*-variables. Since functions of 9-variables or fewer were the only functions considered, it was decided that full enumeration would be performed.

1. Johnson-Trotter Adjacent Transposition Algorithm

It was decided that an algorithm capable of performing all permutations without repetition based on swapping adjacent variable pairs would be the most beneficial in this case. First, the simplicity of the swap would consume minimal processing resources. Second, each permutation inherently has “memory” of the preceding swaps, since no repetitions are permitted. Thus, the program can minimize its footprint in RAM as only two variable orderings are required to exist at any given time: the previous (or original) variable ordering, and the variable ordering generated from this previous ordering.

A non-adjacent transposition algorithm would require the program to maintain explicit knowledge of previous permutations, and would quickly prove unmanageable: a 10-variable function requires a truth table with 1024 entries. Assuming each truth table entry requires a byte of storage (as it does in BDDViewer), 3.7 Gigabytes of RAM would be required to hold the entire set of 3,628,800 permutations. This storage requirement increases factorially, making storage impractical on any system. On the other hand, use of adjacent transposition allows the program to scale up almost indefinitely even on a common personal computer.

The algorithm known as the Johnson-Trotter algorithm [21, 22] met all requirements for this program, and so was adapted for use in BDDViewer. The algorithm assumes that the original permutation is ordered numerically (i.e., the original permutation is considered to be x_1, x_2, … , x_{n-1}, x_n). Each element in the permutation is weighted according to its numerical value; for instance, $x_5 > x_2$. Furthermore, each element is assigned a direction of mobility: either left or right. Initially all elements are set to move left. With these initial conditions set, permutation occurs by having the largest mobile element swap with its neighbor in the direction of mobility. Once an element passes all elements (finding itself on either the far left or far right), the element is no longer mobile. Then the next largest mobile element moves according to the same conditions. When a lower weighted element moves, the direction of mobility for all greater elements is reversed. In most cases, this results in a greater element becoming the mobile element in the next step. The algorithm continues until all permutations have been generated. Coincidentally, once all permutations have been generated, no element can be mobile.

2. Pseudo-Code for Johnson-Trotter

```
for all elements 1 through n: mobility = left.
while there exists a mobile element:
    identify largest mobile element k
    swap element k with neighbor in direction of mobility
    for all elements l > k
        reverse direction of  element l
```

3. Example Permutation

Var1	Var2	Var3	Var4
$< x_1$	$< x_2$	$< x_3$	$< x_4$
$< x_1$	$< x_2$	$< x_4$	$< x_3$
$< x_1$	$< x_4$	$< x_2$	$< x_3$
$< x_4$	$< x_1$	$< x_2$	$< x_3$
$x_4 >$	$< x_1$	$< x_3$	$< x_2$
$< x_1$	$x_4 >$	$< x_3$	$< x_2$
$< x_1$	$< x_3$	$x_4 >$	$< x_2$
$< x_1$	$< x_3$	$< x_2$	$x_4 >$
$< x_3$	$< x_1$	$< x_2$	$< x_4$
$< x_3$	$< x_1$	$< x_4$	$< x_2$
$< x_3$	$< x_4$	$< x_1$	$< x_2$
$< x_4$	$< x_3$	$< x_1$	$< x_2$
$x_4 >$	$x_3 >$	$< x_2$	$< x_1$

Var1	Var2	Var3	Var4
$x_3 >$	$x_4 >$	$< x_2$	$< x_1$
$x_3 >$	$< x_2$	$x_4 >$	$< x_1$
$x_3 >$	$< x_2$	$< x_1$	$x_4 >$
$< x_2$	$x_3 >$	$< x_1$	$< x_4$
$< x_2$	$x_3 >$	$< x_4$	$< x_1$
$< x_2$	$< x_4$	$x_3 >$	$< x_1$
$< x_4$	$< x_2$	$x_3 >$	$< x_1$
$x_4 >$	$< x_2$	$< x_1$	$x_3 >$
$< x_2$	$x_4 >$	$< x_1$	$x_3 >$
$< x_2$	$< x_1$	$x_4 >$	$x_3 >$
$< x_2$	$< x_1$	$x_3 >$	$x_4 >$

Table 8. Application of Johnson-Trotter Algorithm to generate all possible variable orderings of a 4-variable function.

4. Truth Table Application of the Input Variable Permutation Algorithm

Applying the Johnson-Trotter algorithm to the input variables was a trivial process. The algorithm assumes a set composed of ordered integers, which applies easily to the case of input variables. However, the BDDViewer program creates BDDs through the parsing and manipulation of truth tables. No knowledge of the original function or its input variables is used or maintained. In order for a permutation of input variable ordering to be useful, the permutation must correspond in some way to a swapping of truth table values. The nature of truth table manipulations is such that different variable swaps affect the truth table outputs in significantly different ways. Fortunately, the problem is somewhat simplified by the fact that all manipulations are done between adjacent variables.

x_1	x_2	x_3	x_4	f
0	0	0	0	**0**
0	0	0	1	**1**
0	0	1	0	**1**
0	0	1	1	**0**
0	1	0	0	**1**
0	1	0	1	**0**
0	1	1	0	**0**
0	1	1	1	**0**
1	0	0	0	**0**
1	0	0	1	**1**
1	0	1	0	**1**
1	0	1	1	**1**
1	1	0	0	**1**
1	1	0	1	**0**
1	1	1	0	**1**
1	1	1	1	**0**

Table 9. Swapping variables x_1 and x_2 will result in the manipulation of the highlighted truth table entries.

In Table 9, a TT is given for an arbitrary 4-variable function. Swapping variables x_1 and x_2 will require the highlighted values of f to be swapped as well. Specifically $f(\mathrm{v}_4)$ must swap with $f(\mathrm{v}_8)$, $f(\mathrm{v}_5)$ must swap with $f(\mathrm{v}_9)$, $f(\mathrm{v}_6)$ must swap with $f(\mathrm{v}_{10})$, and $f(\mathrm{v}_7)$ must swap with $f(\mathrm{v}_{11})$. Regardless of which input variables are swapped, 2^{n-2} truth table pairs must be swapped for an n-variable function. In other words, each variable swap results in the manipulation of half of the truth table. The following tables will demonstrate how the swapping of different input variable pairs will affect the truth table.

x_1	x_2	x_3	x_4	f
0	0	0	0	**0**
0	0	0	1	**1**
0	0	1	0	**1**
0	0	1	1	**0**
0	1	0	0	**1**
0	1	0	1	**0**
0	1	1	0	**0**
0	1	1	1	**0**
1	0	0	0	**0**
1	0	0	1	**1**
1	0	1	0	**1**
1	0	1	1	**1**
1	1	0	0	**1**
1	1	0	1	**0**
1	1	1	0	**1**
1	1	1	1	**0**

Table 10. Swapping variables x_2 and x_3 in an arbitrary 4-variable function.

x_1	x_2	x_3	x_4	f
0	0	0	0	**0**
0	0	0	1	**1**
0	0	1	0	**1**
0	0	1	1	**0**
0	1	0	0	**1**
0	1	0	1	**0**
0	1	1	0	**0**
0	1	1	1	**0**
1	0	0	0	**0**
1	0	0	1	**1**
1	0	1	0	**1**
1	0	1	1	**1**
1	1	0	0	**1**
1	1	0	1	**0**
1	1	1	0	**1**
1	1	1	1	**0**

Table 11. Swapping variables x_3 and x_4 in an arbitrary 4-variable function.

From Tables 9 through 11, a few patterns become evident. Generally speaking, the farther to the "right" of the truth table that the variable swap takes place, the smaller the "sets" of swapped TT pairs. Specifically, the swap of variables x_3 and x_4 results in four evenly spaced sets of one pair each that must be swapped; swapping variables x_2 and x_3 results in two sets of two pairs each; swapping variables x_1 and x_2 results in one large set of four TT pairs that must be swapped.

For the purpose of developing this algorithm, a variable swap between variables $\underline{x_n}$ and x_{n-1} was considered swap "0." A swap between variables x_{n-2} and x_{n-1} was swap "1." A swap between variables x_1 and x_2 was considered swap "n-1." By defining these swaps in terms of an integer "z" the truth table manipulations can be defined mathematically.

Generally, the following is true:

- Each variable swap will require 2^{n-2} TT pair swaps.
- Variable swap z will have 2^z TT pair swaps in each set, where $0 \le z < n$.
- The first set of TT pair swaps will occur at $f(v_{2^z})$.
- The spacing between sets of TT pair swaps will be 2^{z+1}.
- The spacing between TT pairs will be 2^z .

The above allows one to manipulate a truth table given the swapping of any two adjacent input variables. Since only adjacent input variables will ever be swapped in the Johnson-Trotter algorithm, the above definition is all that is necessary to produce all truth table permutations for a given Boolean function.

5. Pseudo-Code for Truth Table Manipulation

given variable swap number z for function of size n:

```
num_swaps = $2^{n-2}$

pair_index = pair_spacing = set_size = $2^z$

set_spacing = $2^{z+2}$

for i = 0; i < num_swaps

    for j = 0; j < set_size

        swap TT.index(first_pair_index + j) and
            TT.index(first_pair_index + pair_spacing + j)

    pair_index = pair_index + set_spacing
```

C. SUMMARY

In this chapter, the design of BDDViewer was discussed. BDDViewer automatically generates a minimum sized BDD when provided with a function's truth table. Two novel algorithms on BDD generation and truth table manipulation were detailed. The next chapter will discuss the general findings of select samplings of the BDDs of known bent functions that were identified with the aid of this software.

V. DISCOVERIES

With a custom-built, self-contained application capable of receiving a Boolean function, determining the minimum BDD realization, and outputting the results in a graphical format, it became possible to look at a variety of bent functions in a form that has likely never been previously considered. With such a large selection of bent functions to choose from, the difficulty lay in choosing the specific functions to be investigated. For instance, for 6-variables, there exist over five billion bent functions!

To simplify the process somewhat, functions were chosen by their relative ease of construction. A primary reason for this was due to the fact that although all bent functions have been enumerated for $n \leq 8$, very few resources exist that provide a large repository for the known functions.

One benefit of choosing bent functions that are easily constructed is that these functions also tended to exhibit useful properties. This is likely a consequence of the general rule that human designed functions tend to have simple BDDs. Thus well-known bent function constructions also showed relatively simple BDDs. This helped make analysis particularly easy. The following sections will serve to summarize the useful properties discovered of the BDDs of these bent functions.

A. DISJOINT QUADRATIC FUNCTIONS

DQFs were defined in Chapter II, section C. They are easy to construct, since they consist of disjoint monomials and their BDDs are minimum when the variables of the tree have the same ordering as the algebraic normal form. The DQFs of 2-variables, 4-variables, 6-variables, and 8-variables were all investigated, and their BDDs are compiled in Appendix B. Due to the large number of diagrams, the reader is directed to this appendix for all diagrams, so as not to overextend the size of this chapter. Note that red edges correspond with a “0” decision and blue edges correspond with a “1.” Likewise, a red terminal node represents 0 and a blue terminal node represents 1.

# of Variables	# of Non-Terminal Nodes
2	2
4	6
6	10
8	14

Table 12. Number of Non-Terminal Nodes in the BDD of DQFs.

From the BDDs in Appendix B, one finds that the number of non-terminal nodes in a DQF follows a distinct pattern. Specifically, each increment in function size results in 4 additional non-terminal nodes. Recall that both bent functions and DQFs must have an even number of variables.

Theorem 1: The number of non-terminal nodes in a DQF is 2*n*-2 [17].

In the base case, n = 2. This is the basic AND operation on x_1x_2. For a 2-variable AND function, two nodes must exist, since the terminal node of "1" is dependent on the both variables also being 1.

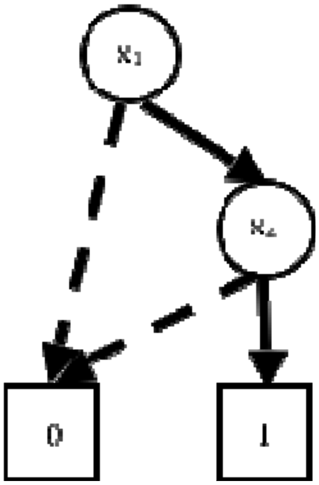


Figure 14. DQF Order 2, AKA the AND Function of two variables.

In order to generate DQFs of higher order, a smaller DQF is simply concatenated by XOR with another AND function. For instance, a DQF of order 4 is x_1x_2 concatenated with x_3x_4, resulting in $x_1x_2 \oplus x_3x_4$. This concatenation results in the addition of two additional AND BDDs as sub-graphs of the original DQF. The first sub-graph stems from the 0 terminal of the original graph, and the second sub-graph stems from the 1 terminal of the original graph.

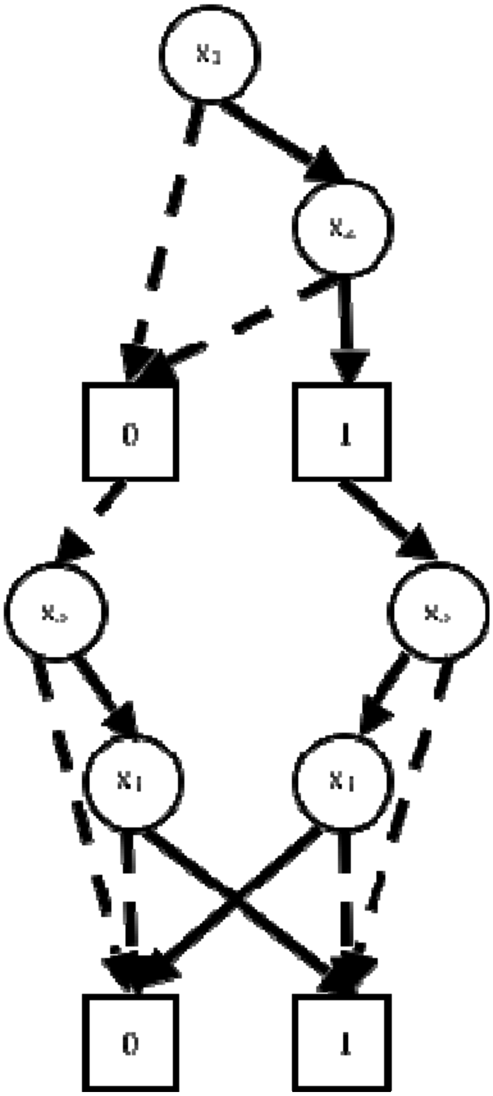


Figure 15. DQF Order 4 generated from concatenation of two disjoint AND functions.

The use of two sets of "terminal nodes" in Figure 15 only serves to illustrate each sub-graph's relation to the original AND function of Figure 14. Ignoring the intermediate "terminal nodes" in the mid section of the graph, note that none of the nodes associated with the x_3 and x_4 variables are isomorphic. Thus, Figure 15 represents a proper ROBDD. Note also that construction of a 6-variable DQF would occur in the same way: with complementary AND sub-graphs appended to each terminal node, or four additional nodes. Hence, the total number of non-terminal nodes in a DQF of order n is $2n$ -2.

B. SYMMETRIC BENT FUNCTIONS

Symmetric bent functions were first discussed in Chapter II, Section C. As a reminder, a bent function f is symmetric if

$$f(x_1, x_2, ..., x_n) = \sum_{1 \le i < j \le n} x_i x_j \oplus c \sum_{i=1}^{n} x_i \oplus d.$$

This thesis focuses primarily on the general case of symmetric bent functions, for which coefficients c and d are 0. Note that coefficient d is relatively uninteresting, since it simply complements the function, which has no impact on the BDD with the exception of inverting each terminal node. Since a c coefficient of 1 serves to invert each input variable, it has the effect of simply inverting each edge. This also does not impact the general structure of the BDD. Thus, the general function is considered adequate for the analysis that follows.

The condensed truth table of the general symmetric bent function is

# of Variables Set to 1	*f*
0	0
1	0
2	1
3	1
4	0
5	0
6	1
7	1
8	0
…	…

Table 13. Condensed Truth Table of a Symmetric Bent Function.

Note that the condensed truth table follows a specific pattern. That is, the as the number of variables set to 1 increases, the output follows the repeating pattern “0-0-1-1.” This is due to the structure of the function, which is composed of the XOR of all possible quadratic monomials. Thus, the number of monomials that are equivalent to 1 is the number of ways two objects can be chosen from *k* without repetition. This can be illustrated by the combination function

$$\binom{k}{2},$$

where *k* is the number of variables set to 1. When the combination function results in an even number, the output *f* must be 0, since an even number of monomials is set to 1. When the combination function results in an odd number, the output *f* must be 1, since an odd number of monomials are set to 1. The proof that incrementing *n* results in a pattern of even-even-odd-odd is left to the reader.

Once again, the reader is directed to Appendix C for a comprehensive collection of BDDs of symmetric bent functions.

# of Variables	# of Non-Terminal Nodes
2	2
4	8
6	16
8	24

Table 14. The Number of Non-Terminal Nodes in the BDDs of Symmetric Bent Functions.

With the exception of the difference between a 4-variable symmetric bent function and a 2-variable symmetric bent function, each increase in size of the symmetric bent functions results in 8 additional nodes to the size of the BDD. Like the DQF, a pattern quickly becomes evident, but in the case of symmetric bent functions, the pattern is somewhat more remarkable.

Theorem 2: The number of non-terminal nodes in a symmetric bent function is 4*n*-8 for *n* ≥ 4 [17].

The linear pattern in the number of nodes of the BDDs of symmetric bent functions is due to the nature of its condensed truth table. As discussed in the previous section, the condensed truth table is composed of a repeating set of four values. This memory causes the edges of the BDD to loop back on themselves as for every four variables set to 1. Because of this, the maximum width of the BDD at any given level must be four nodes. Since each increment in bent function size requires the addition of two variables, or two levels, this is responsible for the increase in 8 nodes shown in the previous section.

To help illustrate this point, a modified, partial BDD of a symmetric function follows. Note that the tree in Figure 16 does not follow the conventions of ROBDD, nor does it represent a proper bent function because it represents an odd number of variables (though it is symmetric).

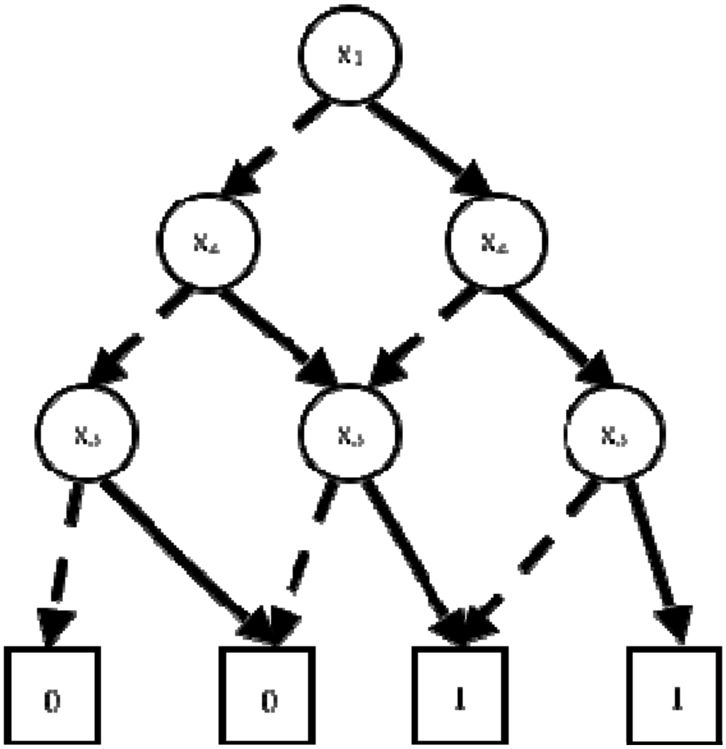


Figure 16. Partial Symmetric Bent Function Example.

In Figure 16, the four condensed truth table values are represented. Note that, just as in the condensed truth table, each value corresponds to the number of variables set to 1. The leftmost terminal 0 represents the case for which no input variables are set to 1. The second terminal 0 represents the case for which one input variable is set to 1. The leftmost terminal 1 represents the case for which two input variables are set to 1, and the final terminal represents the case for which three input variables are set to 1. As shown in the previous section, this pattern repeats, and will be reflected in the BDD.

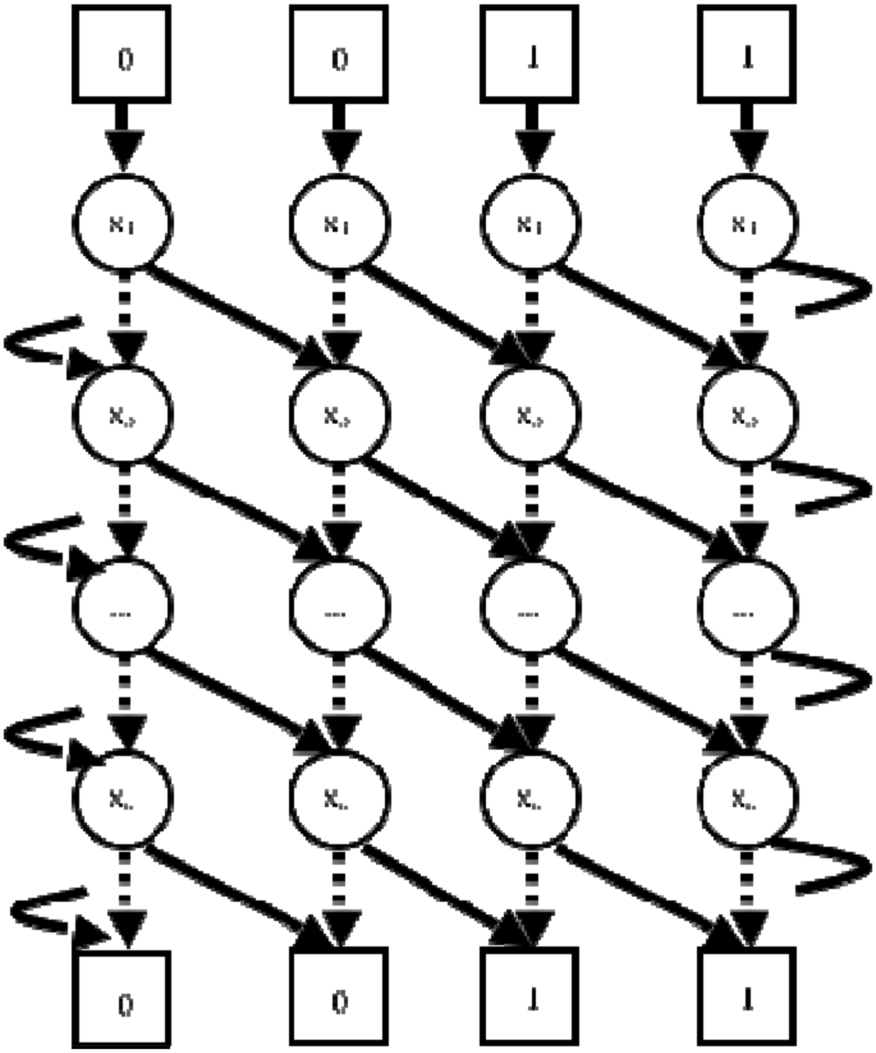


Figure 17. The mid-section of a symmetric bent BDD.

Figure 17 demonstrates that the mid-section of a symmetric bent BDD necessarily has four nodes per level, each node corresponding with its aligned condensed truth table value. Thus, the BDD of a symmetric bent function of *n*-variables has approximately 4*n* nodes. However, the structure of an ROBDD results in some reductions at the top and bottom of each graph. From Figure 16 it is evident that the root level can only have one node, the second level has only two nodes, and the third level has three. This reduces the size of the total BDD by 6 nodes. Furthermore, from Figure 17 it is evident that the leftmost node x_n and the third node x_n are isomorphic and will be skipped entirely, reducing the total size of the BDD by another two nodes. Thus, the ROBDD of a symmetric bent function has $4n - 8$ non-terminal nodes for $n \geq 4$. Once again, the reader is directed to Appendix C for examples of the structure of symmetric bent functions for up to 8-variables.

C. HOMOGENEOUS BENT FUNCTIONS OF ORDER SIX AND ALGEBRAIC DEGREE THREE

Homogeneous functions are those for which all monomials have the same number of variables. Disjoint Quadratic Functions and the general form of the symmetric bent functions are both homogeneous functions in that they are quadratic. Specifically, all monomials have exactly two variables.

This next section analyzes homogeneous cubic bent functions of 6-variables. That is, all monomials have exactly three variables. This particular subset of bent functions was chosen for a few reasons: they are well known; there are only 30 of them, which makes for a naturally small sampling for analysis; and they are interesting in that they are the highest order bent functions of *n*-variables for which their algebraic degree is *n*/2 [1].

In order to analyze these homogeneous bent functions, each was run through the BDDViewer program. The shape and number of nodes of their minimum BDDs were recorded, as well as the number of nodes in the diabolical variable ordering. This process concluded in a surprising result.

Observation: All minimum sized BDDs of homogeneous functions on 6-variables and degree 3 have the same general structure and number of nodes, and each diabolical BDD has the same maximum number of nodes. Specifically, each function has 20 non-terminal nodes in the minimum case, and 24 non-terminal nodes in the maximum case.

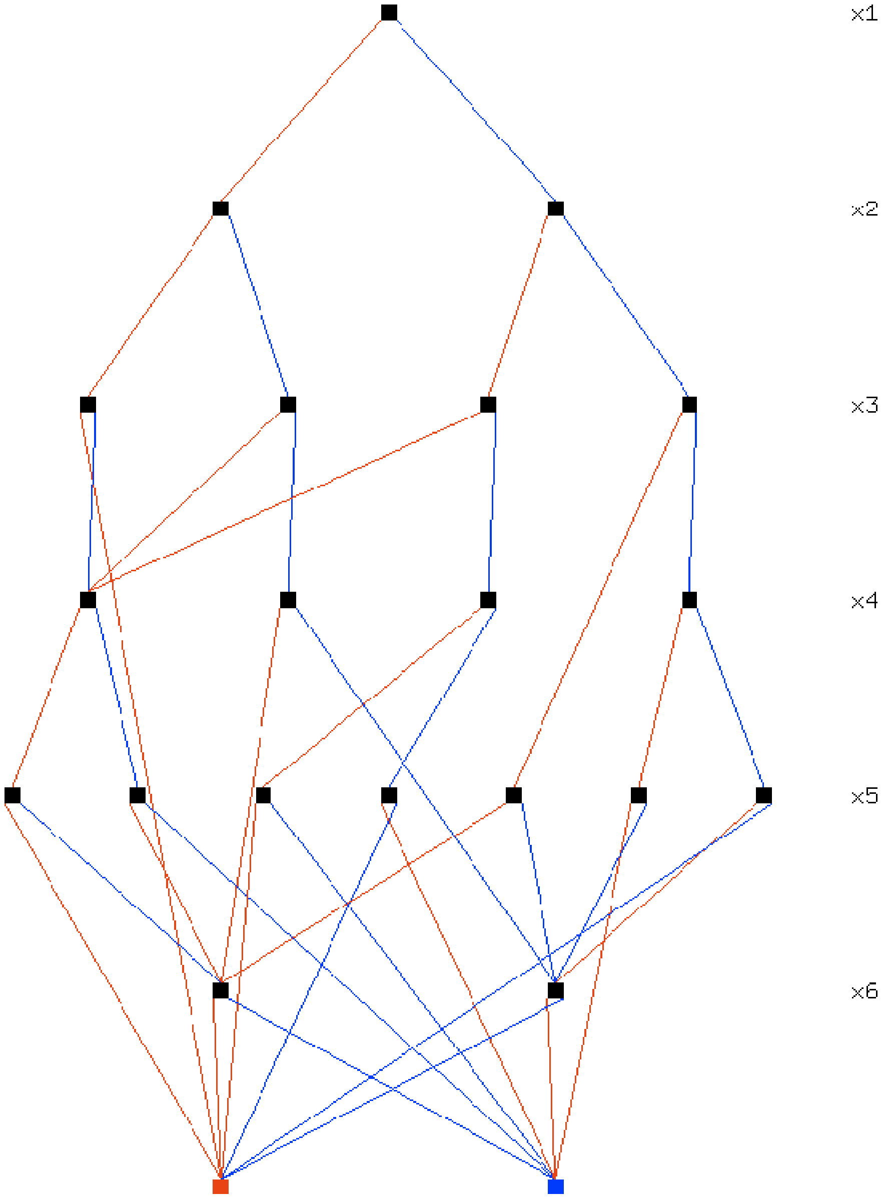


Figure 18. $x_1x_2x_3 \oplus x_1x_2x_5 \oplus x_1x_2x_6 \oplus x_1x_3x_4 \oplus x_1x_3x_5 \oplus x_1x_4x_5 \oplus x_1x_4x_6 \oplus x_1x_5x_6 \oplus x_2x_3x_4 \oplus x_2x_3x_6 \oplus x_2x_4x_5 \oplus x_2x_4x_6 \oplus x_2x_5x_6 \oplus x_3x_4x_5 \oplus x_3x_4x_6 \oplus x_3x_5x_6$

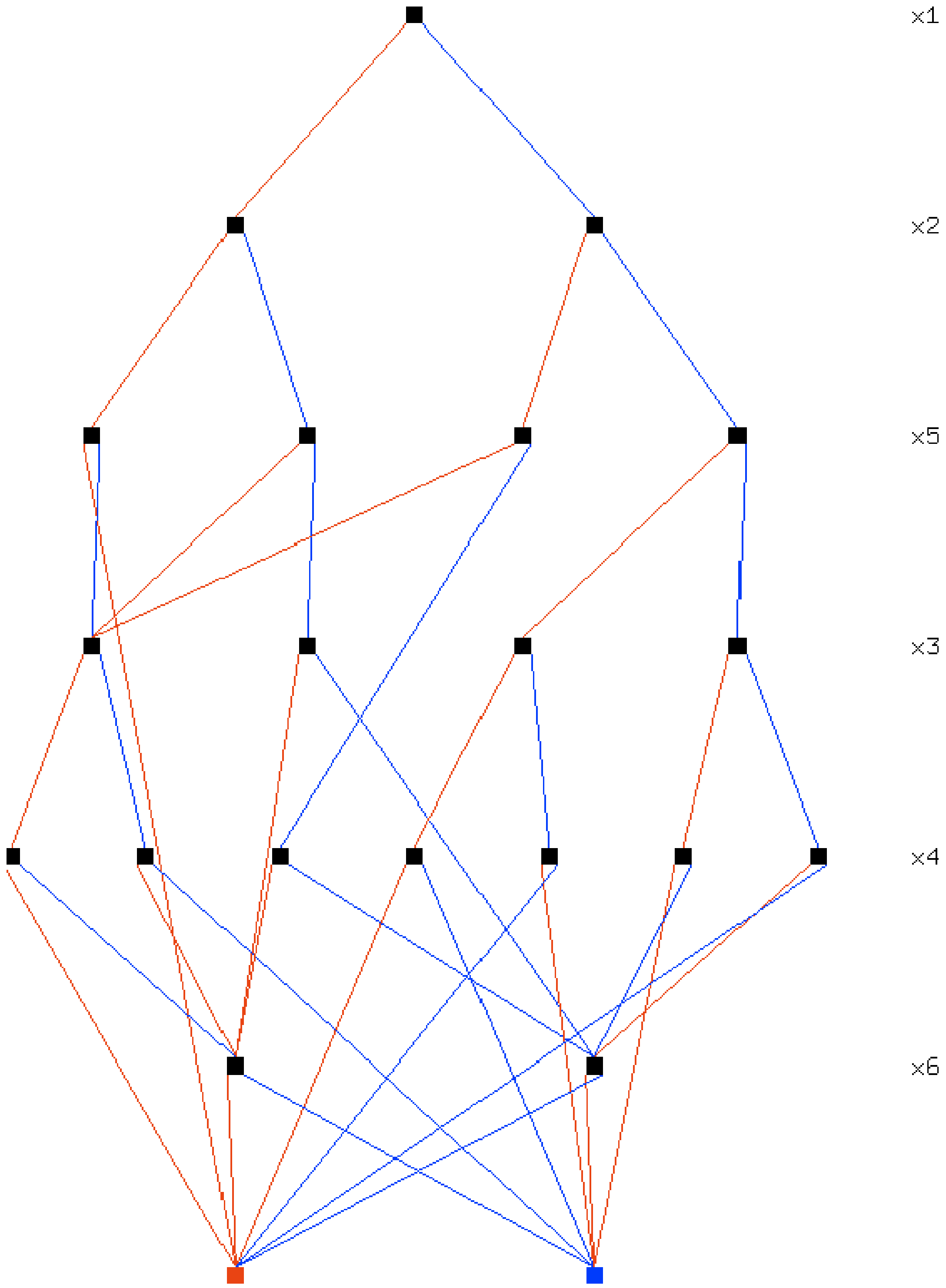


Figure 19. $x_1x_2x_3 \oplus x_1x_2x_4 \oplus x_1x_2x_5 \oplus x_1x_3x_4 \oplus x_1x_3x_6 \oplus x_1x_4x_5 \oplus x_1x_4x_6 \oplus x_1x_5x_6 \oplus x_2x_3x_4 \oplus x_2x_3x_5 \oplus x_2x_3x_6 \oplus x_2x_4x_6 \oplus x_2x_5x_6 \oplus x_3x_4x_5 \oplus x_3x_5x_6 \oplus x_4x_5x_6$

Figures 18 and 19 are minimum BDDs of two different homogeneous bent functions. Note that Figure 18 has a variable ordering of x_1-x_2-x_3-x_4-x_5-x_6 and Figure 19 has a variable ordering of x_1-x_2-x_5-x_3-x_4-x_6. Also, note that each BDD has the same structure. That is, in addition to having the same number of nodes, it also has the same number of nodes per level. However, nodes are not necessarily connected by the same edges. It is also important to remind the reader that a BDD does not necessarily have only one minimum variable ordering. As a specific example, the symmetric functions have the exact same BDD regardless of variable ordering.

Although identical general structure of the BDDs was an interesting result, it was not particularly surprising. It was assumed that, much like the BDDs of the DQFs and the symmetric bent functions, the recurring structure was a result of the structure of the function itself. That is, DQFs show a similar pattern due to their homogeneous nature, so it appeared this was the case for homogeneous functions of degree 3 for 6-variable functions as well.

However, one surprising result did occur. BDDs were discovered between two separate homogeneous functions that were identical, not only in structure but also in edges. Figures 20 and 21 reveal two functions for which this trait was discovered. Of course, the two BDDs have different variable orderings, with Figure 20 having ordering x_1-x_2-x_5-x_3-x_4-x_6 and Figure 21 having ordering x_1-x_4-x_6-x_2-x_3-x_5.

Theorem 3: Two functions are P-equivalent iff those two functions have identical BDDs for distinct variable orderings.

The proof of this is trivial. As shown in Chapter IV, the BDD of a function can be described entirely by its truth table, with no knowledge of the underlying ANF or variable ordering. Thus, if two functions have identical BDDs, they must have identical truth tables, and must therefore be identical functions. Likewise, changing the variable ordering of a BDD implies a permutation of the truth table. Thus, if changing the variable ordering of a BDD results in the truth table of another function, those two functions must be P-equivalent. Also, if two functions are P-equivalent, by definition that means they have identical truth tables after some permutation of the input variables. If they have identical truth tables, they will have identical BDDs. Thus, two functions are P-Equivalent iff those two functions have identical BDDs for distinct variable orderings.

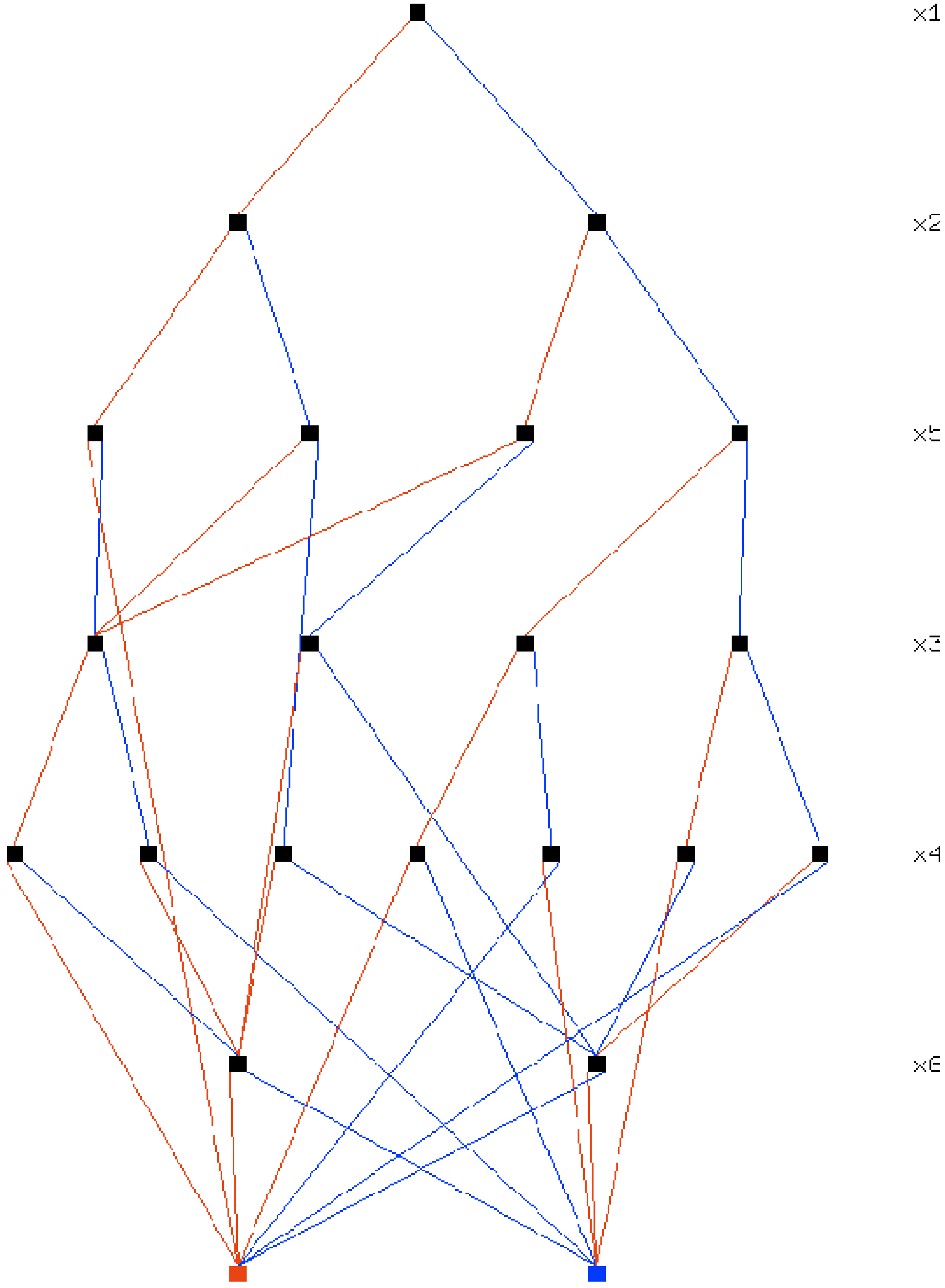


Figure 20. $x_1x_2x_3 \oplus x_1x_2x_4 \oplus x_1x_2x_5 \oplus x_1x_3x_4 \oplus x_1x_3x_5 \oplus x_1x_3x_6 \oplus x_1x_4x_6 \oplus x_1x_5x_6 \oplus x_2x_3x_4 \oplus x_2x_3x_6 \oplus x_2x_4x_5 \oplus x_2x_4x_6 \oplus x_2x_5x_6 \oplus x_3x_4x_5 \oplus x_3x_5x_6 \oplus x_4x_5x_6$

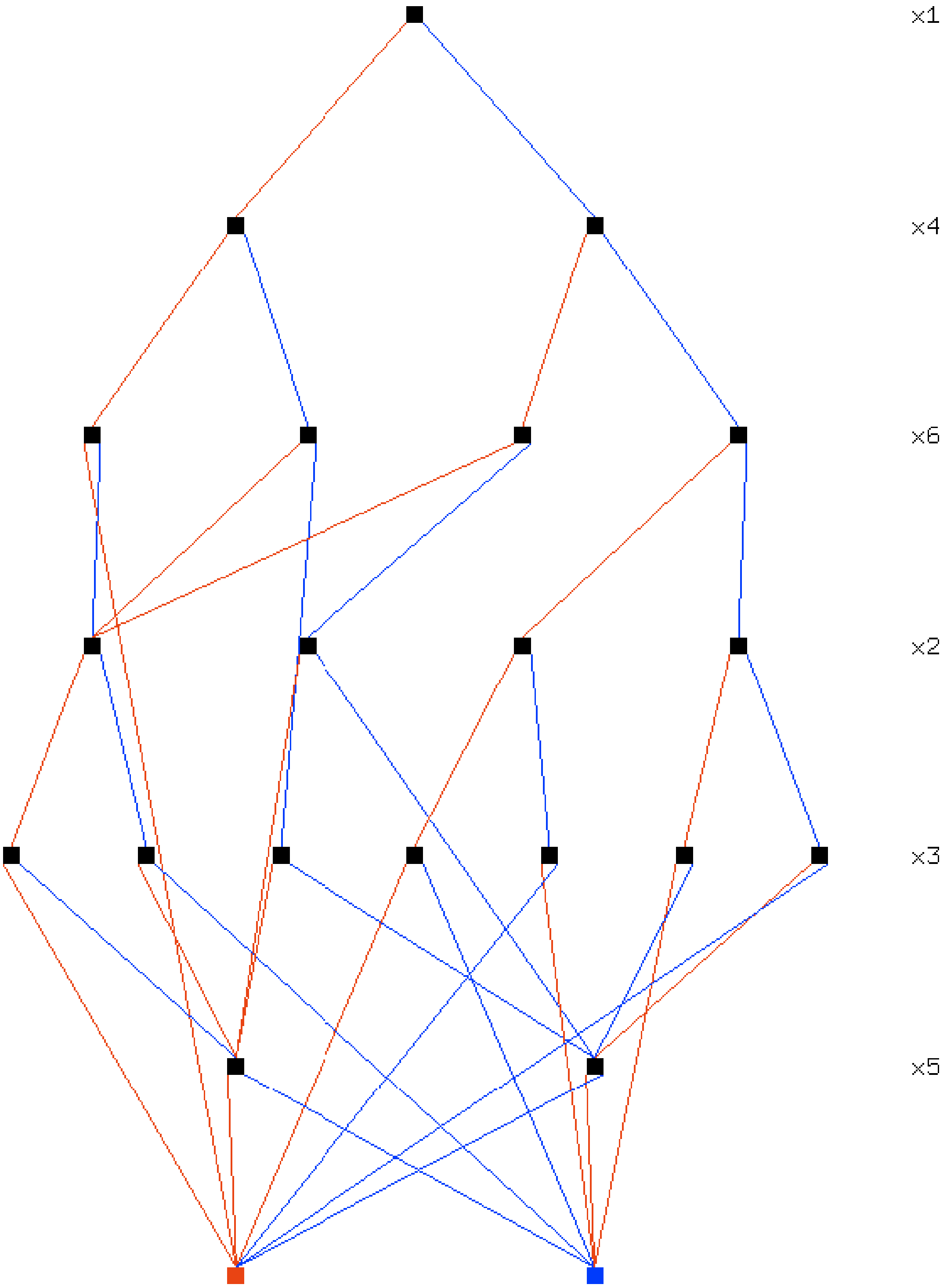


Figure 21. $x_1x_2x_3 \oplus x_1x_2x_4 \oplus x_1x_2x_5 \oplus x_1x_2x_6 \oplus x_1x_3x_4 \oplus x_1x_3x_5 \oplus x_1x_4x_6 \oplus x_1x_5x_6 \oplus x_2x_3x_4 \oplus x_2x_3x_6 \oplus x_2x_4x_5 \oplus x_2x_5x_6 \oplus x_3x_4x_5 \oplus x_3x_4x_6 \oplus x_3x_5x_6 \oplus x_4x_5x_6$

From Theorem 3, it was discovered that all 30 homogeneous cubic functions of 6-variables are P-equivalent. Running each function through all possible BDD permutations revealed that all functions shared identical BDDs for some variable ordering. Though this was proved through enumeration, all 30 BDDs are not provided in this thesis due to the redundancy in the graphs.

D. AFFINE CLASSES

In Chapter II, affine classes were defined as being the set of bent functions derived from a root bent function XORed with any affine function. For some root function *f* and some affine function $\ell_{a,c}$, $f \oplus \ell_{a,c}$ has the effect of complementing each sub-function under x_{a_i} for which $a_i = 1$ when *c* = 0, or for which $a_i = 0$ when *c* = 1.

x_1	x_2	x_3	f	$f \oplus \ell_{a,c}$	$\ell = x_2$
0	0	0	0	0	0
0	0	1	0	0	0
0	1	0	1	**0**	1
0	1	1	0	**1**	1
1	0	0	1	1	0
1	0	1	0	0	0
1	1	0	0	**1**	1
1	1	1	1	**0**	1

Table 15. The effect of an affine function on an arbitrary 3-variable function.

Table 15 demonstrates the effect of an affine function when XORed with a root function *f*. Note that, in this particular case, the affine function is x_2. The truth table is complemented only for the values for which $x_2 = 1$. The truth table is unaffected for the values for which $x_2 = 0$. As described in Chapters III and IV, the truth table can be parsed to represent sub-functions, which are further used to generate the sub-graphs of the BDD.

The complement of a sub-function does not affect the number of isomorphic sub-graphs, so the number of nodes in the BDD remains unchanged. Instead, the edges between nodes must correspond to the complemented terminal nodes, as they represent the output of the sub-function. Thus the edges of the BDD are affected, but not the overall structure of the BDD, where the structure is considered to be the total number of non-terminal nodes and the number of nodes per level.

Appendix B has many examples of the affect of affine functions on DQFs of various orders. Note that while the interconnection of edges is affected, the structure is not. Further examples of the affect of affine functions on some functions can be seen in Appendix D, which contains a general collection of BDDs of a variety of functions not analyzed in depth in this thesis.

Conjecture: The minimum BDDs of all the functions in an affine class have the same structure.

If XORing a root function and an affine function has the effect of complementing some of the sub-graphs of the root function, it follows that the number of nodes at each level of the BDD is unaffected.

E. SUMMARY

In this chapter, the BDDs of several bent functions were investigated. The BDDs of DQFs and symmetric bent functions were revealed to have predictable structures and complexities. Homogeneous bent functions of order 6 and degree 3 were shown to be P-equivalent, which led to the realization that P-equivalent functions have identical BDDs for some distinct variable ordering. Finally, the functions in the affine class of a bent function were shown to share the same general structure and complexity of the root function. The next chapter summarizes the findings further, and offers suggestions for future research.

THIS PAGE INTENTIONALLY LEFT BLANK

VI. CONCLUSIONS AND FUTURE WORK

A. CONCLUSIONS

The purpose of this research was to investigate the unique characteristics of the binary decision diagrams of Boolean bent functions. In particular, the focus of analysis was on a small sampling of easily constructible functions in the form of disjoint quadratic functions, symmetric bent functions, and the homogeneous bent functions on 6-variables. Though these functions are well known, it is believed that they have not been observed from the perspective of their binary decision diagrams previously.

From the research, it was found that both disjoint quadratic functions and symmetric bent functions have minimum BDDs with predictable sizes. Furthermore, the minimum BDD size is linear with *n*, the number of variables, which is a desirable characteristic for the implementation of these functions in either software or hardware.

Although it was known previously that the homogeneous bent functions on 6-variables were P-equivalent, the findings of this thesis helped identify that the P-equivalency of functions can be identified from the structure of their BDDs. Furthermore, the research indicates that all of the functions within an affine class have BDDs of the same general structure and size. Since affine classes are quite large, a single structure can be implemented within strict limitations of hardware or software that may be reconfigured into a variety of unique functions.

Though the research focused on the BDDs of particular bent functions and their characteristics, it should not be overlooked that doing so required the design of a software application capable of generating, permuting, and displaying these functions quickly. Although BDDViewer is not robust in terms of the number of operations that can be performed on a BDD, it does provide a simple and fast method of computing and displaying the minimum BDD of a known function. The program is not limited to bent functions, and can be used to describe any Boolean function.

B. FUTURE WORK

1. On BDDs and Bent Functions

The process of discovering bent functions for $n \geq 10$ is still relatively unrefined. All bent functions are known for $n \leq 8$ primarily due to enumeration. As has been discussed in earlier chapters, the exact number of bent functions for *n*-variables is not known, and the potential number of bent functions is broadly bounded. Because of the high degree of uncertainty, the discovery of bent functions for $n \geq 10$ is usually accomplished by selecting a group of variables with high potential for high-nonlinearity and enumerating the sample. Use of genetic algorithms to identify new bent functions is also a possibility, though it is unlikely to aid in identification of anything but small fractions of the total solution space at a time, and without the benefit of allowing researchers to verify that the entire solution space has been exhausted.

Constructing a BDD requires knowledge of a function's truth table, and so research into bent functions is inherently limited by the known set of functions. Furthermore, research of the known functions is constricted by the fact that very few, if any, repositories of the known bent functions exist. However, researchers should not be discouraged. Only a few families of bent functions were analyzed for the purpose of this thesis, but unique properties were nonetheless identified.

An obvious starting point for future research would be to investigate the relationships of the BDDs of homogeneous bent functions for $n \geq 8$. These functions are not all P-Equivalent, as was the case for $n = 6$, but the properties of those functions may yet prove interesting.

In Rothaus' original paper [1], several families of bent functions were identified. Only the few deemed immediately relevant to this thesis were defined in Chapter II. Since the families of bent functions are grouped together because they share specific mathematical properties, the results may prove intriguing.

Finally, in this thesis, it was shown that the affine classes shared BDD of the same general size and shape. Just as the function that results from XORing a bent function with an affine function is known to be bent, it is also true that XORing a bent function with another bent function is also bent [2]. How would the BDD be affected by this operation? Would one BDD dominate the other? How might the size or shape be affected?

Bent functions are not necessarily the only functions that merit investigation, though they are valuable because they are rare and they are highly non-linear. Most bent functions are modified to incorporate other cryptographically desirable properties before being implemented in a cryptographic system. For instance, the cryptographically desirable property of balancedness is in direct contention with bent functions because bent functions are not balanced. What are the properties of the BDDs of balanced and highly non-linear functions that are more likely to be implemented in a cryptographic system?

2. BDDViewer

Although BDDViewer's functionality sufficed for the purposes of this thesis, a lot of work can still be put into the features and efficiency of the program. The BDDs were generated by representing the truth table as a "string" data type, which may not be the most efficient way of representing the data. Furthermore, the graphical output of each BDD was done with OpenGL, with which the author had no previous experience. Experienced programmers are likely to identify several flaws in the current code revision. It should be noted, however, that the current version of the program only draws one BDD at a time. Therefore, the amount of computation time devoted to drawing the BDD is minimal.

Another area of focus could be the porting of code to a Windows operating system. BDDViewer currently works only on the Macintosh platform, although only standard C++ and OpenGL libraries were used. Porting the code would allow a much larger audience to have access to the application, and would likely not require much effort since no Mac-centric or Windows-centric libraries should be necessary.

Finally, several features could be modified. Currently, the window size is fixed, since at the time of writing the program, it was known that only functions of 10-variables or less would be investigated. However, even for 8 or 9-variables, the size of the current window becomes cramped and difficult to read.

The method for minimizing BDDs could also be updated. The computation time required to check all possible BDD permutations for $n \leq 10$ was low enough that total enumeration was viable, but doing so for $n > 10$ is impractical. An optional heuristic method could be implemented for larger n values, though doing so may prevent the program from finding the true minimum. The work of Gunther and Drechsler [18] may provide a good starting point for this modification.

Finally, since functions generally have several variable orderings that result in minimum-sized BDDs, the program could be updated to capture several or all minimum BDDs and save the results to a file. In its current form, only the first minimum tree is displayed.

APPENDIX A: CODE

A. TEXT-BASED TREE DATA STRUCTURE

```
/*
 *  Tree.h
 *  BDD
 *
 *  Created by Neil Schafer on 2/13/09.
 *  Last modified: 2/24/09.
 *  Copyright 2009 Naval Postgraduate School. All rights reserved.
 *
 */

#include <string>
#include <iostream>
#include <math.h>
using namespace std;

struct node
{
    string myValue;           // The TT element that corresponds to 
this node

    int myLevel;              // The level in the tree that the node is 
located

    bool traversed;           //Tells the print function if this node 
has already been seen

    node * child0;            // The child node of this element for an 
input of 0

    node * child1;            // The child node of this element for an 
input of 1

    node * myParent;          // The parent of the node
};

class BDDTree
{
    string myTT;                    // Truth Table of the entire tree
    string list_of_sub[100];        // List of subfunctions
    node * list_of_del[100];        // List of deleted functions
    node myRoot;                    // Root Nodes
    node * currentNode;             // Used in tree traversal
    int myOrder;        // The number of variables
    int mySize;         // The number of nodes
    int myWidth;        // The highest number of nodes in one level
    int maxLevel;
    int myDepth;        // The deepest level of the tree

    public:
```

```
    BDDTree(string TT);
    bool addSub(string subF);
    int size();
    bool deleted(node * child);
    node * attachChild(string subF);
    void initialize();
    void newChild(node * parent, int childNum, string value);
    void printTree();
    void Destruct();
    ~BDDTree();
};

int BDDTree::size()
{
    return mySize;
}

BDDTree::BDDTree(string TT)
//Creates the first node of the tree
{
    myTT = TT;
    list_of_sub[0] = TT;

    for(int i= 1; i <100; i++)
    {
      list_of_sub[i] = "-1";
      list_of_del[i] = NULL;
    }

    myRoot.myValue = TT;
    myRoot.myLevel = 0;
    myRoot.traversed = false;
    myRoot.child0 = NULL;
    myRoot.child1 = NULL;
    myRoot.myParent = NULL;
    myOrder = log(TT.size())/log(2);
    currentNode = &myRoot;
    mySize = 0;
    myWidth = 0;
    myDepth = 1;
    maxLevel = 0;
}

bool BDDTree::deleted(node * child)
{
    int i = 0;
    while(list_of_del[i] != NULL)
    {
      if(child == list_of_del[i]) return true;
      i++;
    }

    list_of_del[i] = child;
    return false;
}
```

```
bool BDDTree::addSub(string subF)
{
    int i = 0;
    while(list_of_sub[i] != "-1")
    {
      if(subF == list_of_sub[i]) break;
      i++;
    }

    if(subF == list_of_sub[i]) return false;

    list_of_sub[i] = subF;
    return true;
}

node * BDDTree::attachChild(string subF)
{
    node * temp = currentNode;
    node * child;

    if(temp->myValue == subF)
    {
      currentNode = &myRoot;
      return temp;
    }

    if(temp->child0)
    {
      currentNode = temp->child0;
      child = attachChild(subF);
      if(child) return child;
    }

    if(temp->child1)
    {
      currentNode = temp->child1;
      child = attachChild(subF);
      if(child) return child;
    }

    return NULL;
}



void BDDTree::initialize()
// Initializes a tree to the TT provided recursively.
{
    node * temp = currentNode;
    string TT_Low, TT_High, TT;
    TT = currentNode->myValue;
    mySize++;

    if(currentNode->myValue.size() > 1)
    //if the node's TT is large enough to be broken in half again,
    //make some new nodes
    {
      TT_Low = TT.substr(0, TT.size()/2);
      TT_High = TT.substr(TT.size()/2, TT.size()/2);

      if (currentNode == &myRoot)
      {
            while (TT_Low == TT_High && TT_Low.size() >0)
            {
                  addSub(TT_Low);
                  myRoot.myLevel = log(this-
>myTT.size()/TT_Low.size())/log(2);
                  myRoot.myValue = TT_Low;
                  TT_Low = TT_Low.substr(0, TT_Low.size()/2);
                  TT_High = TT_High.substr(TT_High.size()/2,
TT_High.size()/2);
            }
      //    if(TT_Low.size() == 1) return;
            if(myRoot.myValue.size()==1) return;
      }

      while(TT_Low.size()>1 && TT_Low.substr(0, TT_Low.size()/2) ==
TT_Low.substr(TT_Low.size()/2, TT_Low.size()/2))
      {
            addSub(TT_Low);
            TT_Low = TT_Low.substr(0, TT_Low.size()/2);
      }
      while(TT_High.size()>1 && TT_High.substr(0, TT_High.size()/2) ==
TT_High.substr(TT_High.size()/2, TT_High.size()/2))
      {
            addSub(TT_High);
            TT_High = TT_High.substr(0, TT_High.size()/2);
      }


      if(this->addSub(TT_Low))
      {
            currentNode = temp;
            newChild(currentNode, 0, TT_Low);
            currentNode = temp->child0;
            this->initialize();
      }
      else
      {
            currentNode = &myRoot;
            temp->child0 = attachChild(TT_Low);
      }

      if(this->addSub(TT_High))
      {
            currentNode = temp;
            newChild(currentNode, 1, TT_High);
            currentNode=temp->child1;
            this->initialize();
      }
      else
      {
            currentNode = &myRoot;
            temp->child1 = attachChild(TT_High);
      }

    }

    currentNode = &myRoot; //after initialization is complete
                                //reset currentNode to tree root.
}

void BDDTree::newChild(node * parent, int childNum, string value)
// Creates a child node for a given parent. The node corresponds to the
TT value given
{
    node * temp;
    if(childNum == 0)
    {
      temp = new(node);
      temp->myValue = value;
      temp->myLevel = log(this->myTT.size()/value.size())/log(2);
      temp->traversed = false;
      temp->child0 = NULL;
      temp->child1 = NULL;
      parent->child0 = temp;
    }

    else if(childNum == 1)
    {
      temp = new(node);
      temp->myValue = value;
      temp->myLevel = log(this->myTT.size()/value.size())/log(2);
      temp->traversed = false;
      temp->child0 = NULL;
      temp->child1 = NULL;
      parent->child1 = temp;
    }

    if (temp->myLevel > maxLevel)
    {
      maxLevel = temp->myLevel;
      myDepth++;
    }
}



void BDDTree::printTree()
//Recursively lists all the nodes in the tree
{
    node * temp = currentNode;
    if(temp->traversed) return;
    else temp->traversed = true;

    if (currentNode==&myRoot)
    {
      cout << "Tree Depth: " << myDepth << endl;
    //      cout << "Tree Width: " << myWidth << endl;
      cout << "Tree Size:  " << mySize << endl;
      cout << "Level " << '\t' << "Node Value " <<  '\t' << "0 Child"
<< '\t' << "1 Child" << endl;
    }
    cout << temp->myLevel << "\t\t" << temp->myValue << '\t';

    if(temp->child0) cout << temp->child0->myValue << '\t';
    else cout << "NULL" << '\t';

    if(temp->child1) cout << temp->child1->myValue << endl;
    else cout << "NULL" << endl;


    if(temp->child0)
    {
      currentNode = temp->child0;
      printTree();
    }

    currentNode = temp;

    if(temp->child1)
    {
      currentNode = temp->child1;
      printTree();
    }

    currentNode = &myRoot;
}


void BDDTree::Destruct()
//Recursively frees the tree from memory.
//This code is probably a bit sloppy.
{
    node * temp = currentNode;
    if(temp->child0)
    {
      if(!deleted(temp->child0))
      {
            currentNode = temp->child0;
            Destruct();
    //            temp->child0 = NULL;
      }
    }

    if(temp->child1)
    {
      if(deleted(temp->child1))
      {
            currentNode = temp->child1;
            Destruct();
    //            temp->child1 = NULL;
      }
    }

    if(temp != &myRoot)
    {
      delete temp;
      currentNode = NULL;
      temp = NULL;
    }
}

BDDTree::~BDDTree()
{
    Destruct();
}
```

B. GRAPHICS-BASED TREE DATA STRUCTURE

```
/*
 *  Tree.h
 *
 *  Defines 1 Class and 2 structs:
 *
 *  class BDDTree:
 *  Instantiated with a boolean Truth Table (input as a string).
Creates a Reduced Order
 *  Binary Decision Diagram tree from the previously supplied Truth
Table.
 *  To reveal the BDD, the printTree() function MUST be called. This
prints
 *  a text version of the BDD to the console window, and is also
necessary to
 *  generate the "coordinate system" for graphical display.
 *
 *
 *  struct node
 *  a general node structure that is typically used when creating
trees.
 *  this does have a "traversed" boolean value, because I couldn't
think of any
 *  better ways to avoid printing nodes that share multiple ancestors
multiple times
 *  since the printTree() works recursively...
 *  Also includes "coordinates" for itself and its children.
 *  coordinates[0] is a unique identifier for the node. Coordinates[1]
is its
 *  level in the tree. It's up to the user to correlate that somehow to
a graphical
 *  output. Since the tree is generated from "left to right", the
coordinates[0] for a
```

```
 *  given level will always be lower on the left and will alway
increase (not necessarily
 *  linearly!) as you go to the right. That might help...
 *
 *  struct CoordinateHolder
 *  a cheap hack that holds a two dimensional array so that it can be
returned to
 *  the calling program.
 *
 *  Notes:
 *  I should probably move all the functions declared in the Tree
definition
 *  except for those described above to the private section...
 *
 *
 *
 *  Sloppy stuff to work on:
 *  (1) Figure out a better coordinate system.
 *  (2) Get rid of the parent node.
 *
 *  Created by Neil Schafer on 2/13/09.
 *  Last modified: 4/7/09.
 *  Copyright 2009 Naval Postgraduate School. All rights reserved.
 *
 */

#include <string>
#include <iostream>
#include <math.h>
using namespace std;

struct node
{
    string myValue;                 // The TT subfunction that
corresponds to this node

    int myLevel;              // The level in the tree that the node is
located

    bool traversed;                 // Tells the print function if this
node has already been printed

    int myCoord[2];                 // This node's placement. Helps
display graphically
    int child0Coord[2];             // Helps link nodes graphically
    int child1Coord[2];             // Helps link nodes graphically
    node * child0;                  // The child node of this element
for an input of 0

    node * child1;                  // The child node of this element
for an input of 1

    node * myParent;          // The parent of the node (this is sloppy
                              // find a way to get rid of it.
};
```

```
struct CoordinateHolder
// nasty
{
    int myCoordinates[300][6];
};

class BDDTree
{
    string myTT;                    // Truth Table of the entire tree
    string list_of_sub[300];  // List of subfunctions
    node * list_of_del[300];        // List of deleted functions
    CoordinateHolder treeCoordinates;     //Lists of coordinates for
graphical display
    node myRoot;                    // Root Nodes
    int ZeroThenOne;
    node * currentNode;             // Used in tree traversal
    int myOrder;        // The number of variables (don't think I use
this one either)
    int mySize;               // The number of nodes
    int myWidth;        // The highest number of nodes in one level
(DOESN'T WORK)
    int maxLevel;       // do I use this?
    int myDepth;        // The deepest level of the tree (STUPID)
    bool addSub(string subF); //adds a subfunction to the stored list
of functions
    bool deleted(node * child);
    void initialize();                          // The most important
function
    node * attachChild(string subF);      // returns a pointer to the
node with
                                                      // the provided
subfunction value

    void newChild(node * parent, int childNum, string value);     //
creates a new node

    public:
    CoordinateHolder returnCoordinates();
    BDDTree(string TT);
    int size();
    int LeftRight();
    void printTree();
    void Destruct();
    ~BDDTree();
};

CoordinateHolder BDDTree::returnCoordinates()
// returns the two-dimensional array of coordinates to the user
{
    return treeCoordinates;
}


int BDDTree::size()
```

```
// returns the size of the finished tree. Useful for minimization
{
    return mySize;
}

BDDTree::BDDTree(string TT)
//Sets up the Tree.
{
    myTT = TT;
    ZeroThenOne = -1;

    for(int i= 0; i <300; i++)
    {
      list_of_sub[i] = "-1";
      list_of_del[i] = NULL;
      treeCoordinates.myCoordinates[i][0] = 0;
      treeCoordinates.myCoordinates[i][1] = 0;
      treeCoordinates.myCoordinates[i][2] = 0;
      treeCoordinates.myCoordinates[i][3] = 0;
      treeCoordinates.myCoordinates[i][4] = 0;
      treeCoordinates.myCoordinates[i][5] = 0;
    }

    list_of_sub[0] = TT;

    myRoot.myValue = TT;
    myRoot.myLevel = 0;
    myRoot.myCoord[0] = 0;
    myRoot.myCoord[1] = 0;
    myRoot.child0Coord[0] = 0;
    myRoot.child0Coord[1] = 0;
    myRoot.child1Coord[0] = 0;
    myRoot.child1Coord[1] = 0;
    myRoot.traversed = false;
    myRoot.child0 = NULL;
    myRoot.child1 = NULL;
    myRoot.myParent = NULL;
    myOrder = log(TT.size())/log(2);
    currentNode = &myRoot;
    mySize = 0;
    myWidth = 0;
    myDepth = 1;
    maxLevel = 0;

    this->initialize();
}

bool BDDTree::deleted(node * child)
// Sloppy way to check if a shared node has already been
// deleted so I don't delete it twice...
// if the child has been previously deleted, returns "true"
// if the child is free to be deleted, returns "false"
{
    int i = 0;
    while(list_of_del[i] != NULL)
    {
      if(child == list_of_del[i]) return true;
      i++;
    }

    list_of_del[i] = child;
    return false;
}

bool BDDTree::addSub(string subF)
// similar to the "deleted" function
// stores a list of known nodes so that extra
// versions of shared nodes aren't inadvertantly created
// recursively.
// returns false if node already exists
// returns true if the node is safe to be added.
{
    int i = 0;
    while(list_of_sub[i] != "-1")
    {
      if(subF == list_of_sub[i]) return false;
      i++;
    }

    list_of_sub[i] = subF;
    return true;
}

node * BDDTree::attachChild(string subF)
// recursively traverses the tree until a node is found that matches
// the subfunction given.
// returns the address of the node.
{
    node * temp = currentNode;
    node * child;

    if(temp->myValue == subF)
    // node found, returns address of that node
    // resets the "current node" counter
    {
      currentNode = &myRoot;
      return temp;
    }

    if(temp->child0)
    // checks down the left side of the tree
    // for the subfunction
    {
      currentNode = temp->child0;
      child = attachChild(subF);
      if(child) return child;
    }

    if(temp->child1)
    // checks down the right side of the tree
    {
      currentNode = temp->child1;
      child = attachChild(subF);
      if(child) return child;
    }


    //node not found on this path
    return NULL;
}



void BDDTree::initialize()
// Generates the ROBDD recursively
{
    node * temp = currentNode;
    string TT_Low, TT_High, TT;
    TT = currentNode->myValue;
    mySize++;

    if(currentNode->myValue.size() > 1)
    // if the node's TT is large enough to be broken in half again,
    // make some new nodes
    // otherwise, this must be a terminal node
    {
      TT_Low = TT.substr(0, TT.size()/2); // generates the "0" decision
sub function
      TT_High = TT.substr(TT.size()/2, TT.size()/2); //generates the
"1" decision subf

      if (currentNode == &myRoot)
      // the root is a special case, as it might not necessarily be at
level 0 (x1).
      // it might need to move down the tree if nodes are found to
      // be redundant.
      {
      //    myRoot.myCoord[0] = mySize;
      //    myRoot.myCoord[1] = myRoot.myLevel+1;
            while (TT_Low == TT_High && TT_Low.size() >0)
            {
                  addSub(TT_Low);
                  myRoot.myLevel = log(this-
>myTT.size()/TT_Low.size())/log(2);
                  myRoot.myValue = TT_Low;
                  TT_Low = TT_Low.substr(0, TT_Low.size()/2);
                  TT_High = TT_High.substr(TT_High.size()/2,
TT_High.size()/2);
            }
            myRoot.myCoord[0] = mySize;
            myRoot.myCoord[1] = myRoot.myLevel+1;
            if(myRoot.myValue.size()==1) return;
      }

      while(TT_Low.size()>1 && TT_Low.substr(0, TT_Low.size()/2) ==
TT_Low.substr(TT_Low.size()/2, TT_Low.size()/2))
      {
            addSub(TT_Low);
            TT_Low = TT_Low.substr(0, TT_Low.size()/2);
      }
      while(TT_High.size()>1 && TT_High.substr(0, TT_High.size()/2) ==
TT_High.substr(TT_High.size()/2, TT_High.size()/2))
      {
            addSub(TT_High);
            TT_High = TT_High.substr(0, TT_High.size()/2);
      }


      if(this->addSub(TT_Low))
      {
            currentNode = temp;
            newChild(currentNode, 0, TT_Low);
            temp->child0Coord[0]=mySize+1;
            temp->child0Coord[1] = log(this-
>myTT.size()/TT_Low.size())/log(2)+1;
            currentNode = temp->child0;
            currentNode->myCoord[0] = mySize+1;
            currentNode->myCoord[1] = currentNode->myLevel+1;
            currentNode->child0Coord[0] = 0;
            currentNode->child0Coord[1] = 0;
            currentNode->child1Coord[0] = 0;
            currentNode->child1Coord[1] = 0;

            this->initialize();
      }
      else
      {
            currentNode = &myRoot;
            temp->child0 = attachChild(TT_Low);
            temp->child0Coord[0] = temp->child0->myCoord[0];
            temp->child0Coord[1] = temp->child0->myCoord[1];
      }

      if(this->addSub(TT_High))
      {
            currentNode = temp;
            newChild(currentNode, 1, TT_High);
            temp->child1Coord[0]=mySize+1;
            temp->child1Coord[1] = log(this-
>myTT.size()/TT_High.size())/log(2)+1;
            currentNode=temp->child1;
            currentNode->myCoord[0] = mySize+1;
            currentNode->myCoord[1] = currentNode->myLevel+1;
            currentNode->child0Coord[0] = 0;
            currentNode->child0Coord[1] = 0;
            currentNode->child1Coord[0] = 0;
            currentNode->child1Coord[1] = 0;
            this->initialize();
      }
      else
      {
            currentNode = &myRoot;
            temp->child1 = attachChild(TT_High);
            temp->child1Coord[0] = temp->child1->myCoord[0];
            temp->child1Coord[1] = temp->child1->myCoord[1];
      }

    }

    currentNode = &myRoot; //after initialization is complete
                                 //reset currentNode to tree root.
}

void BDDTree::newChild(node * parent, int childNum, string value)
// Creates a child node for a given parent. The node corresponds to the
TT value given
{
    node * temp;
    if(childNum == 0)
    {
      temp = new(node);
      temp->myValue = value;
      temp->myLevel = log(this->myTT.size()/value.size())/log(2);
      temp->traversed = false;
      temp->child0 = NULL;
      temp->child1 = NULL;
      parent->child0 = temp;
    }

    else if(childNum == 1)
    {
      temp = new(node);
      temp->myValue = value;
      temp->myLevel = log(this->myTT.size()/value.size())/log(2);
      temp->traversed = false;
      temp->child0 = NULL;
      temp->child1 = NULL;
      parent->child1 = temp;
    }

    if (temp->myLevel > maxLevel)
    {
      maxLevel = temp->myLevel;
      myDepth++;
    }
}



void BDDTree::printTree()
//Recursively lists all the nodes in the tree
{
    int i=0;
    node * temp = currentNode;
    if(temp->traversed) return;
    else temp->traversed = true;

    if (currentNode==&myRoot)
    {
      cout << "Tree Size:  " << mySize << endl;
      cout << "Level " << '\t' << "Node Value " <<  '\t' << "0 Child"
<< '\t' << "1 Child" << endl;
    }
    cout << temp->myLevel << "\t\t" << temp->myValue << '\t';
    while(treeCoordinates.myCoordinates[i][0]!=0)
    {
      i++;
    }
    treeCoordinates.myCoordinates[i][0] = temp->myCoord[0];
    treeCoordinates.myCoordinates[i][1] = temp->myCoord[1];
    treeCoordinates.myCoordinates[i][2] = temp->child0Coord[0];
    treeCoordinates.myCoordinates[i][3] = temp->child0Coord[1];
    treeCoordinates.myCoordinates[i][4] = temp->child1Coord[0];
    treeCoordinates.myCoordinates[i][5] = temp->child1Coord[1];



    if(temp->child0) cout << temp->child0->myValue << '\t';
    else cout << "NULL" << '\t';

    if(temp->child1) cout << temp->child1->myValue << endl;
    else cout << "NULL" << endl;


    if(temp->child0)
    {
      currentNode = temp->child0;
      if((temp->child0->myValue == "0") && (ZeroThenOne == -1))
ZeroThenOne = 1;
      else if ((temp->child0->myValue == "1") && (ZeroThenOne == -1))
ZeroThenOne = 0;
      printTree();
    }

    currentNode = temp;

    if(temp->child1)
    {
      currentNode = temp->child1;
      printTree();
    }

    currentNode = &myRoot;
}


void BDDTree::Destruct()
//Recursively frees the tree from memory.
//This code is probably a bit sloppy.
```

```
{
    node * temp = currentNode;
    if(temp->child0)
    {
      if(!deleted(temp->child0))
      {
            currentNode = temp->child0;
            Destruct();
    //            temp->child0 = NULL;
      }
    }

    if(temp->child1)
    {
      if(!deleted(temp->child1))
      {
            currentNode = temp->child1;
            Destruct();
    //            temp->child1 = NULL;
      }
    }

    if(temp != &myRoot)
    {
      delete temp;
      currentNode = NULL;
      temp = NULL;
    }
}

int BDDTree::LeftRight()
{
    return ZeroThenOne;
}

BDDTree::~BDDTree()
{
    Destruct();
}
```

C. MAIN: OPENGL AND CONSOLE APPLICATION

```
#include <cstdlib>
#include <GLUT/glut.h>
#include "Tree.h"

using namespace std;

string TT = "0110"; //Global variable so can be reused
CoordinateHolder treeCoords;
int LeftRight;
int order=0;
string minPerm;
```

```
//Functions for tree manipulation
long long factorial(int size);
void permutations(string input, int order, string * key, string *
perms);
string TT_swap(string input, int order, int count);
string keys_swap(string input, int order, int count);
void swap(char & a, char &b);
void createModel(void);


string hex2bin(string hex)
{
    string bin = "";
    for(int i = 0; i<hex.size(); i++)
    {
      switch (hex[i])
      {
            case '0':
            {
                  bin += "0000";
                  break;
            }
            case '1':
            {
                  bin += "0001";
                  break;
            }
            case '2':
            {
                  bin += "0010";
                  break;
            }
            case '3':
            {
                  bin += "0011";
                  break;
            }
            case '4':
            {
                  bin += "0100";
                  break;
            }
            case '5':
            {
                  bin += "0101";
                  break;
            }
            case '6':
            {
                  bin += "0110";
                  break;
            }
            case '7':
            {
                  bin += "0111";
                  break;
            }
            case '8':
            {
                  bin += "1000";
                  break;
            }
            case '9':
            {
                  bin += "1001";
                  break;
            }
            case 'A':
            case 'a':
            {
                  bin += "1010";
                  break;
            }
            case 'B':
            case 'b':
            {
                  bin += "1011";
                  break;
            }
            case 'C':
            case 'c':
            {
                  bin += "1100";
                  break;
            }
            case 'D':
            case 'd':
            {
                  bin += "1101";
                  break;
            }
            case 'E':
            case 'e':
            {
                  bin += "1110";
                  break;
            }
            case 'F':
            case 'f':
            {
                  bin += "1111";
                  break;
            }
      }
    }

    return bin;
}
```

```
void printText(string text)
{
    for(int i = 0; i<text.size(); i++)
    {
      glutBitmapCharacter(GLUT_BITMAP_9_BY_15, text[i]);
    }
}


void RenderScene(void)
{


    glClear(GL_COLOR_BUFFER_BIT);

      int widthCount, width0, width1, level0, level1, index0, index1,
myIndex, value0, value1;
      float xAdjustment;
      float yAdjustment;
      int minPermPosition;
      string level;
      for(int j = 0; j < order+1; j++)
      // finds the number of nodes in each level (there are order+1
levels)
      {
            widthCount = 0;
            for(int m = 0; m < 300; m++)
            //finds the number of nodes at each level
            {
                  if (treeCoords.myCoordinates[m][1] == (j+1))
widthCount++;
                  if (treeCoords.myCoordinates[m][0] == 0) break;
            }

            xAdjustment = 640.0f/(widthCount+1);
            yAdjustment = 750.0f/order;
            minPermPosition = 3*j;

                  glColor3f(0.0f, 0.0f, 0.0f);
                  glRasterPos2f(620.0f, (j)*yAdjustment + 35);
                  level = minPerm.substr(minPermPosition, 2);
                  printText(level);


            for(int n = 0; n < widthCount; n++)
            //prints out those nodes
            {
                  if(j < order)
                  {
                        glColor3f(0.0f, 0.0f, 0.0f);
                        glRectf((n+1)*xAdjustment, 25.0f +
(j)*yAdjustment, (n+1)*xAdjustment+10.0f, 35.0f + (j)*yAdjustment);
```

```
                  }
                  else
                  {
                        if(LeftRight && (n == 0))
                        {
                              glColor3f(1.0f, 0.0f, 0.0f);
                              glRectf((n+1)*xAdjustment, 25.0f +
(j)*yAdjustment, (n+1)*xAdjustment+10.0f, 35.0f + (j)*yAdjustment);
                        }
                        else if (LeftRight && (n == 1))
                        {
                              glColor3f(0.0f, 0.0f, 1.0f);
                              glRectf((n+1)*xAdjustment, 25.0f +
(j)*yAdjustment, (n+1)*xAdjustment+10.0f, 35.0f + (j)*yAdjustment);
                        }
                        else if (!LeftRight && (n == 0))
                        {
                              glColor3f(0.0f, 0.0f, 1.0f);
                              glRectf((n+1)*xAdjustment, 25.0f +
(j)*yAdjustment, (n+1)*xAdjustment+10.0f, 35.0f + (j)*yAdjustment);
                        }
                        else
                        {
                              glColor3f(1.0f, 0.0f, 0.0f);
                              glRectf((n+1)*xAdjustment, 25.0f +
(j)*yAdjustment, (n+1)*xAdjustment+10.0f, 35.0f + (j)*yAdjustment);
                        }
                  }

                  myIndex = 0;
                  width0 = 0;
                  width1 = 0;
                  if(j != order)
                  {
                        for(int p = 0; p < 300; p++)
                        {
                              if(treeCoords.myCoordinates[p][1] ==
(j+1)) myIndex++;
                              if(myIndex == n+1)
                              {
                                    level0 =
treeCoords.myCoordinates[p][3];
                                    value0 =
treeCoords.myCoordinates[p][2];
                                    level1 =
treeCoords.myCoordinates[p][5];
                                    value1 =
treeCoords.myCoordinates[p][4];

                                    index0 = 0;
                                    index1 = 0;
                                    for(int q = 0; q <300; q++)
                                    {

    if(treeCoords.myCoordinates[q][1] == level0)
```

```
                                          {

    if(treeCoords.myCoordinates[q][0] <= value0) index0++;
                                                width0++;
                                          }

    if(treeCoords.myCoordinates[q][1] == level1)
                                          {

    if(treeCoords.myCoordinates[q][0] <= value1) index1++;
                                                width1++;
                                          }
                                    }


                                    //0 Child
                                    glBegin(GL_LINES);
                                          glColor3f(1.0f, 0.0f, 0.0f);
//red

    glVertex2f(myIndex*xAdjustment, 35.0f + j*yAdjustment);

    glVertex2f(index0*640.0f/(width0+1)+5.0f, 25.0f + (level0-
1)*yAdjustment);
                                    glEnd();

                                    //1 Child
                                    glBegin(GL_LINES);
                                          glColor3f(0.0f, 0.0f, 1.0f);
//blue

    glVertex2f(myIndex*xAdjustment+10.0f, 35.0f + j*yAdjustment);

    glVertex2f(index1*640.0f/(width1+1)+5.0f, 25.0f + (level1-
1)*yAdjustment);
                                    glEnd();

                                    break;
                              }
                        }
                  }

            }
      }

      glFlush();

}

void SetupRC(void)
{

    glClearColor(1.0f, 1.0f, 1.0f, 1.0f);
}
```

```
void ChangeSize(GLsizei w, GLsizei h)
{
    GLfloat nRange = 100.0f;
    GLfloat fAspect;

    if(h==0) h= 1;

    fAspect = (GLfloat) w / (GLfloat) h;

    glViewport(0, 0, w, h);

    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();

    gluOrtho2D(0, 640, 800, 0);

    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
}

void idle(void)
{
    createModel();
    glutPostRedisplay();
}

void createModel(void)
{
    string mode;
    char previous;
    long long fact;
    int minSize;
    int maxSize;


    BDDTree * thisTree;
    BDDTree * minTree;
    BDDTree * maxTree;

    cout << "Use previously entered TT? (If no previous entry, default
is 0110) [y/n]: ";
    cin >> previous;
    if(previous == 'n')
    {
      cout << "Enter Truth Table (hex): " ;
      cin >> TT;
      TT = hex2bin(TT);
    }


      cout <<     "Output mode [raw/max/min]: ";
      cin >> mode;

      order = log(TT.size())/log(2);
      fact = factorial(order);
      string myPerms[fact];
      string myKeys[fact];
      string minimumTT;

      if(mode == "min") //look for minimum value
      {
            cout << "Creating permutations" << endl; // debugging
            permutations(TT, order, myKeys, myPerms);

            //Print out the permutations
      /*    for(int i = 0; i< fact; i++)
            cout << myPerms[i] << " " << myKeys[i] << endl;*/

            cout << "Creating trees" << endl; // debugging
            minTree = new BDDTree(myPerms[0]);
            maxSize = minTree->size();
            minSize = minTree->size();
            delete minTree;

            minPerm = myKeys[0];
            minimumTT = myPerms[0];

            int percentage = 0.01*fact;

            //find minimum variation
            for(int i = 1; i < fact; i++)
            {

                  if(i == percentage)
                  {
                        cout << ".";
                        percentage += 0.01*fact;
                  }
                  thisTree = new BDDTree(myPerms[i]);

                  if (thisTree->size() > maxSize) maxSize = thisTree-
>size();

                  if (thisTree->size() < minSize)
                  {
                        minSize = thisTree->size();

                        minPerm = myKeys[i];
                        minimumTT = myPerms[i];
                  }

                  //edit out the following if you don't want to display
the permutations
                  if (thisTree->size() == minSize)
                  {
                        cout << myPerms[i] << " " << minSize << " " <<
myKeys[i] << endl;
                  }

                  delete thisTree;
```

```
        }

    //  cout << "Finished creating trees" << endl; //debugging
        minTree = new BDDTree(minimumTT);

        minTree->printTree();
        treeCoords = minTree->returnCoordinates();
        LeftRight = minTree->LeftRight();

        delete minTree;

        cout << "Minimum permutation is: " << minPerm << endl;
        cout << "Maximum permutation is of size: " << maxSize <<
endl;
    }
    else if(mode == "max") //look for maximum value
    {
        cout << "Creating permutations" << endl;
        permutations(TT, order, myKeys, myPerms);

        //Print out the permutations
    /*  for(int i = 0; i< fact; i++)
        cout << myPerms[i] << " " << myKeys[i] << endl;*/
        cout << "Creating trees" << endl;

        maxTree = new BDDTree(myPerms[0]);
        minSize = maxTree->size();
        maxSize = maxTree->size();
        delete maxTree;

        minPerm = myKeys[0];
        minimumTT = myPerms[0];

        int percentage = 0.01*fact;
        //find minimum variation
        for(int i = 1; i < fact; i++)
        {
            if(i==percentage)
            {
                cout << ".";
                percentage += 0.01*fact;
            }

            thisTree = new BDDTree(myPerms[i]);

            if (thisTree->size() < minSize) minSize = thisTree-
>size();

            if (thisTree->size() > maxSize)
            {
                minPerm = myKeys[i];
                maxSize = thisTree->size();
                minimumTT = myPerms[i];
            }
```

```
                  delete thisTree;
            }

            maxTree = new BDDTree(minimumTT);
            maxTree->printTree();
            treeCoords = maxTree->returnCoordinates();
            LeftRight = maxTree->LeftRight();

            delete maxTree;

            cout << "Maximum permutation is: " << minPerm << endl;
            cout << "Minimum permutation is of size: " << minSize <<
endl;
      }
      else
      {
      //first create non string of variable orderings
            minPerm = "";
                  for(int z = 1; z <= order; z++)
                  {
                        if(z<10)
                        {
                              minPerm += 'x';
                              minPerm += '0'+z;
                              minPerm += ' ';
                        }
                        else
                        {
                              minPerm += "x1";
                              minPerm += '0'+z-10;
                              minPerm += ' ';
                        }
                  }
            thisTree = new BDDTree(TT);
            thisTree->printTree();

            treeCoords = thisTree->returnCoordinates();
            LeftRight = thisTree->LeftRight();

            delete thisTree;
      }
}



int main(int argc, char** argv)
{
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_SINGLE | GLUT_RGBA);
    glutInitWindowSize(640, 800);
    glutCreateWindow("BDD Viewer");
    glutDisplayFunc(RenderScene);
    glutReshapeFunc(ChangeSize);
    glutIdleFunc(idle);
    SetupRC();

    glutMainLoop();

    return 0;
}

long long factorial(int size)
{
    long value = 1;
    for(int i = size; i > 1; i--)
    {
      value = (long)i*value;
    }

    return value;
}

void permutations(string input, int order, string * keys, string *
perms)
{
//Uses the Johnson-Trotter Algorithm

    int swapCount;
    int pair[order+1];
    int direction[order+1];
    long long j, s, q;
    long long counter;
    counter = 0;

    for (int k = 0; k <= order; k++)
    {
      pair[k] = 0;
      direction[k] = 1;
    }


    for(int i = 1; i <= order; i++)
    {
      if(i<10)
      {
            keys[0] += 'x';
            keys[0] += '0'+i;
            keys[0] += ' ';
      }
      else
      {
            keys[0] += "x1";
            keys[0] += '0'+i-10;
            keys[0] += ' ';
      }
    }

    perms[0] = input;
    j = order;
    s = 0;
    counter++;

    do
    {
      q = pair[j] + direction[j];
      if(q < 0)
      {
            direction[j] *= -1;
            j--;
      }
      else if (q == j)
      {
            s++;
            direction[j] *= -1;
            j--;
      }
      else
      {
            if(pair[j] < q) swapCount =order-j + pair[j] - s;
            else  swapCount = order - j + q - s;
            perms[counter] = TT_swap(perms[counter-1], order,
swapCount);
            keys[counter] = keys_swap(keys[counter-1], order,
swapCount);
            counter++;
            pair[j] = q;
            j = order;
            s = 0;
      }
    }
    while(j != 1);
}

string TT_swap(string input, int order, int count)
{
    int num_pairs = pow((double)2, (double)order-2);
    int adj_pairs = pow((double)2, (double)count);
    int small_jump = pow((double)2, (double)count);
    int large_jump = pow((double)2, (double)count+2);
    int index = pow((double)2, (double)count);
    int i = 0;

    while(i < num_pairs)
    {
      for(int j = 0; j < adj_pairs; j++)
      {
            swap(input[index+j], input[index+small_jump+j]);
            i++;
      }
      index += large_jump;
    }

    return input;
```

```
}

void swap(char & a, char &b)
{
    char temp = a;
    a = b;
    b = temp;
}

string keys_swap(string input, int order, int count)
{
    int indexA = input.size()-2-3*count;
    int indexB = indexA - 3;
    if(order < 10)
    {
      swap(input[indexA], input[indexB]);
    }
    return input;
}
```

APPENDIX B: DISJOINT QUADRATIC FUNCTIONS

A. DISJOINT QUADRATIC AFFINE CLASS OF ORDER 2

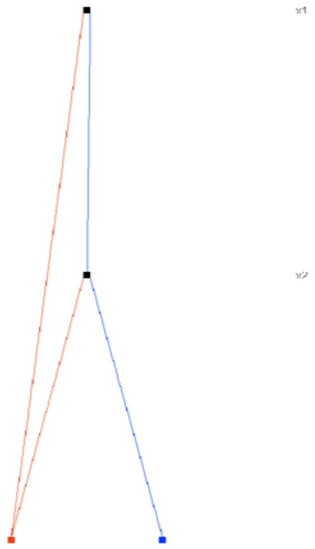


Figure 22. DQF of order 2 (x_1x_2).

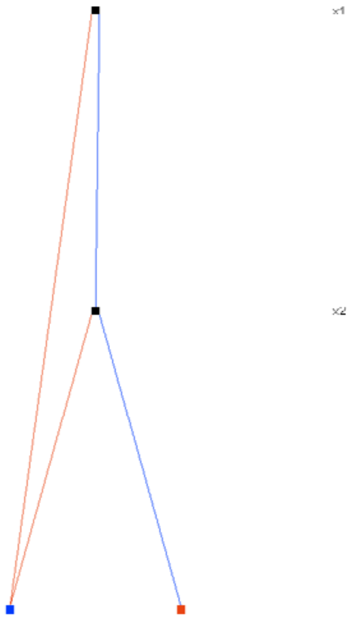


Figure 23. Complement of DQF ($x_1x_2 \oplus 1$).

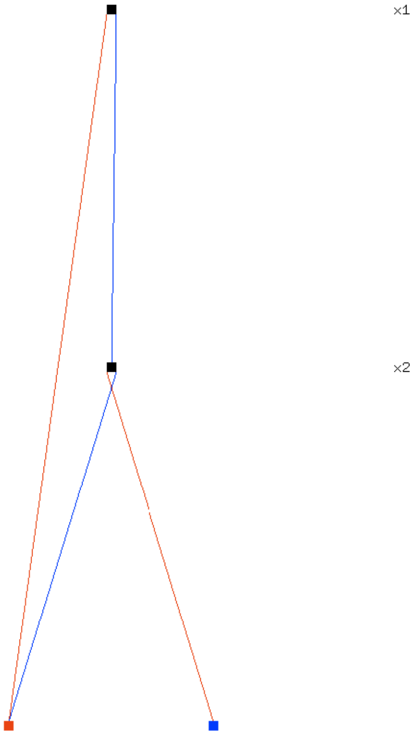


Figure 24. $x_1x_2 \oplus x_1$

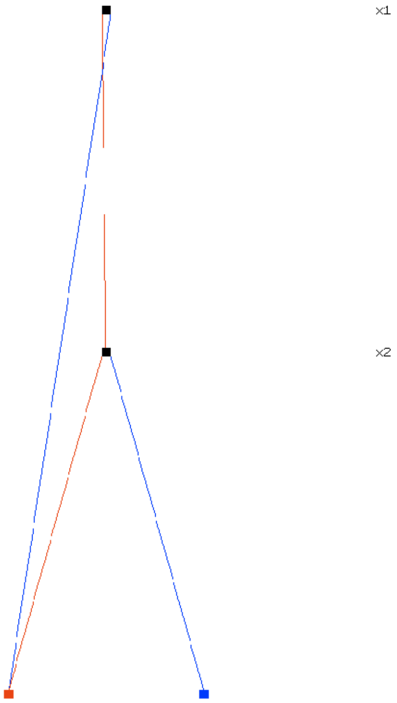


Figure 25. $x_1x_2 \oplus x_2$

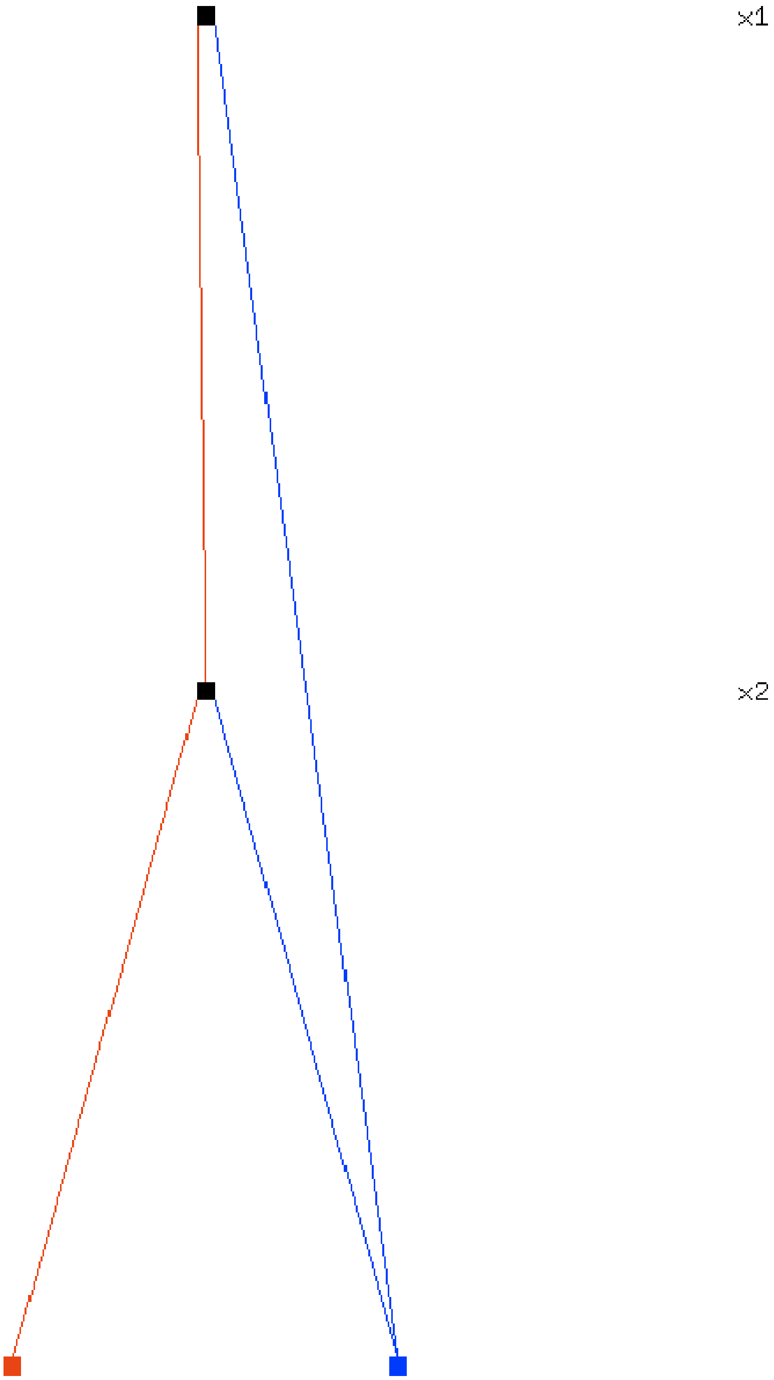


Figure 26. $x_1x_2 \oplus x_1 \oplus x_2$

B. DISJOINT QUADRATIC AFFINE CLASS OF ORDER 4

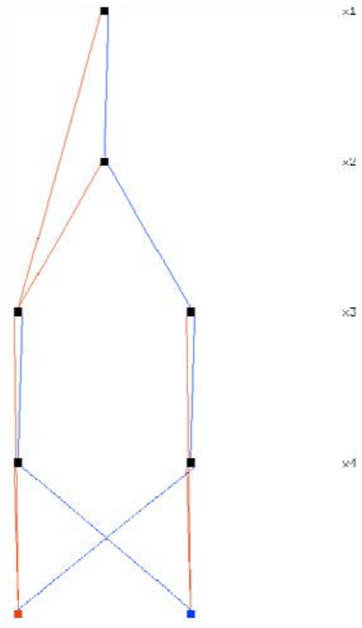


Figure 27. DQF of Order 4 ($x_1x_2 \oplus x_3x_4$).

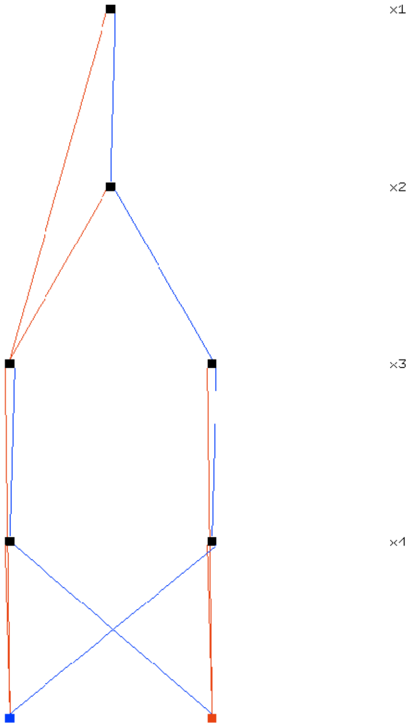


Figure 28. Complement of DQF ($x_1x_2 \oplus x_3x_4 \oplus 1$).

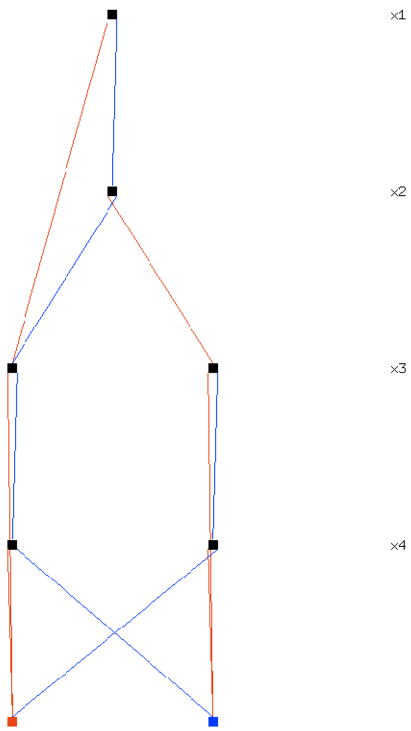


Figure 29. $x_1x_2 \oplus x_3x_4 \oplus x_1$

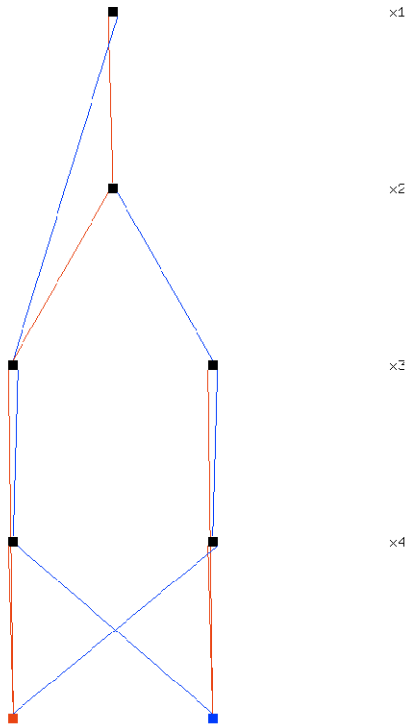


Figure 30. $x_1x_2 \oplus x_3x_4 \oplus x_2$

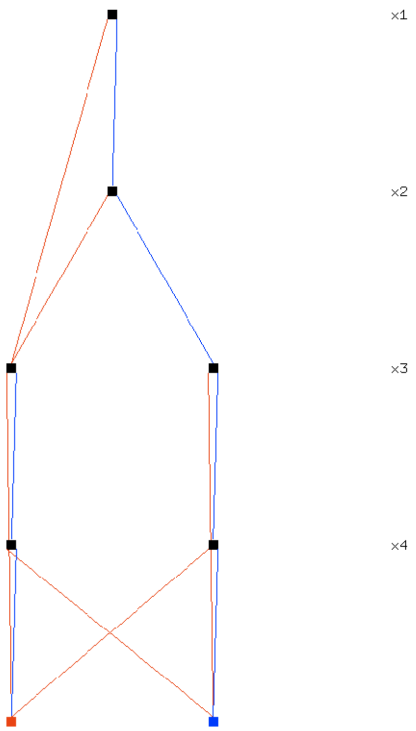


Figure 31. $x_1x_2 \oplus x_3x_4 \oplus x_3$

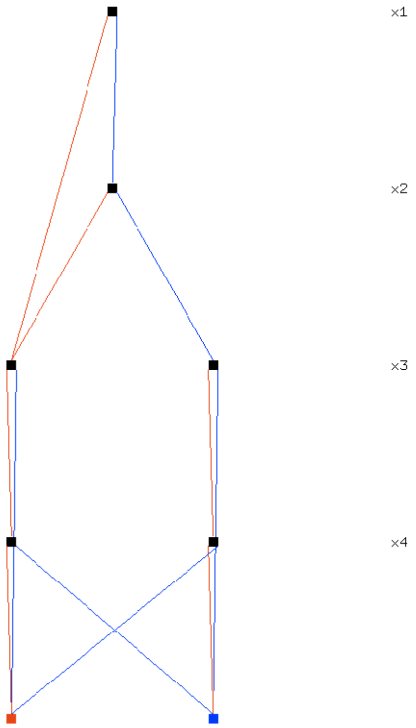


Figure 32. $x_1x_2 \oplus x_3x_4 \oplus x_4$

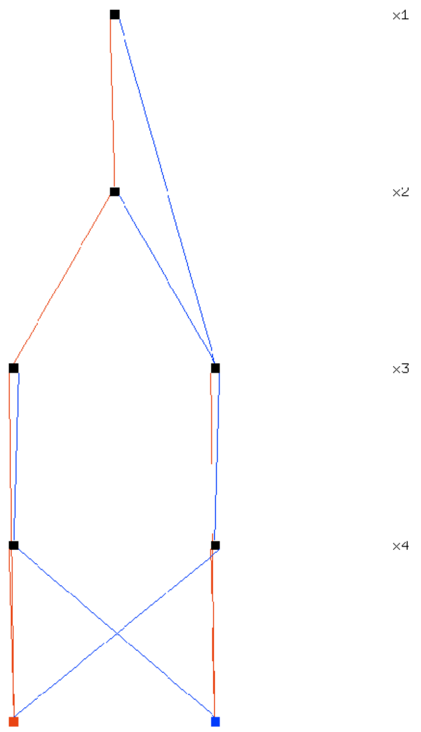


Figure 33. $x_1x_2 \oplus x_3x_4 \oplus x_1 \oplus x_2$

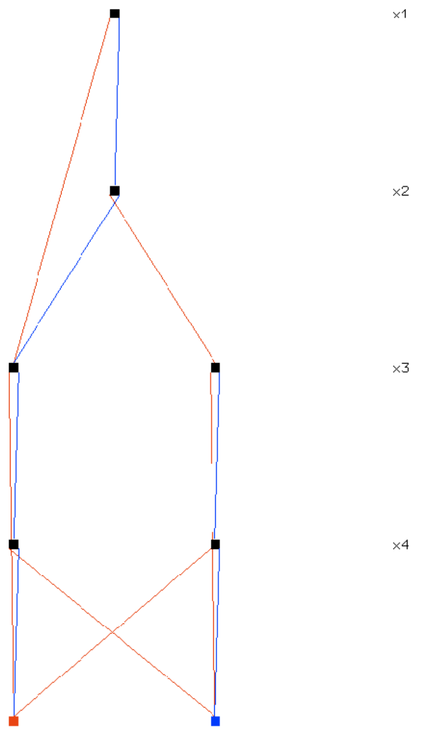


Figure 34. $x_1x_2 \oplus x_3x_4 \oplus x_1 \oplus x_3$

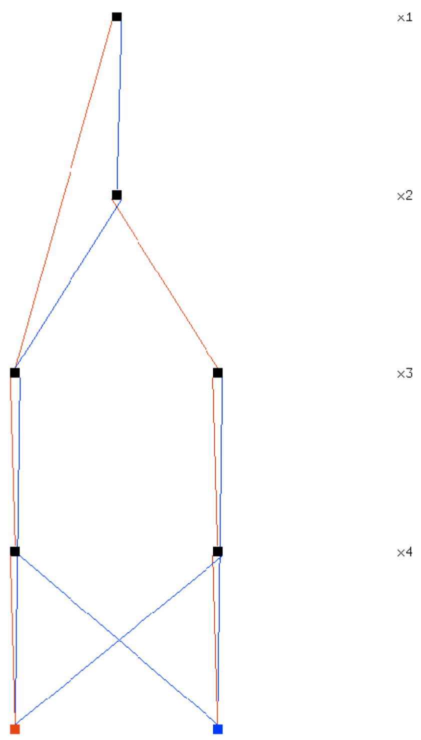


Figure 35. $x_1x_2 \oplus x_3x_4 \oplus x_1 \oplus x_4$

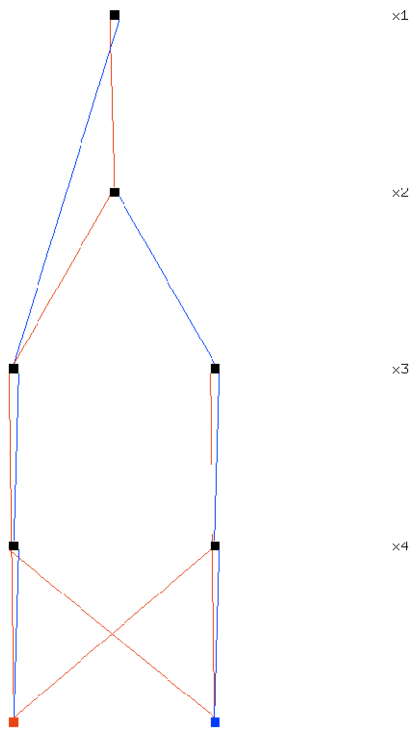


Figure 36. $x_1x_2 \oplus x_3x_4 \oplus x_2 \oplus x_3$

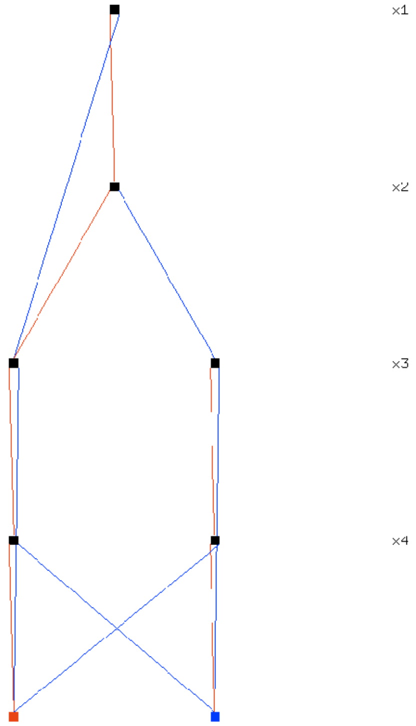


Figure 37. $x_1x_2 \oplus x_3x_4 \oplus x_2 \oplus x_4$

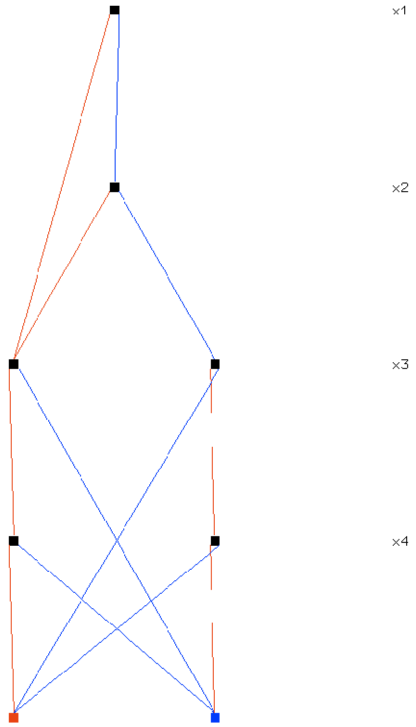


Figure 38. $x_1x_2 \oplus x_3x_4 \oplus x_3 \oplus x_4$

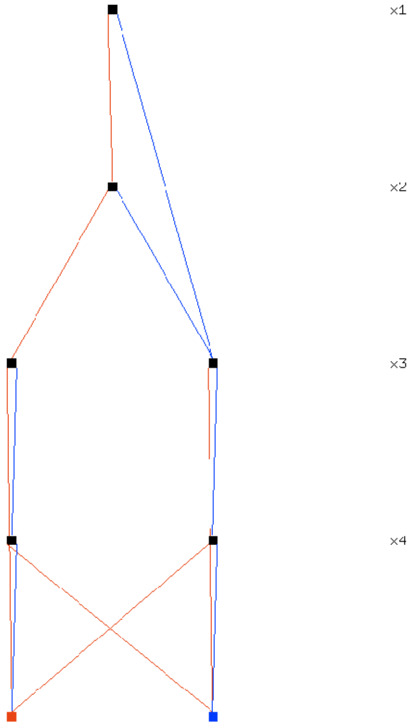


Figure 39. $x_1x_2 \oplus x_3x_4 \oplus x_1 \oplus x_2 \oplus x_3$

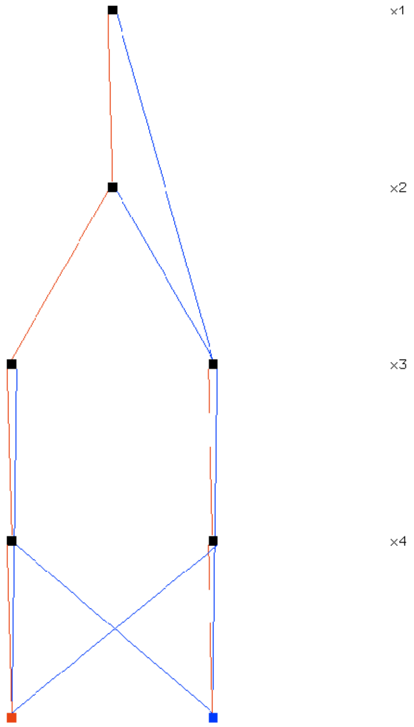


Figure 40. $x_1x_2 \oplus x_3x_4 \oplus x_1 \oplus x_2 \oplus x_4$

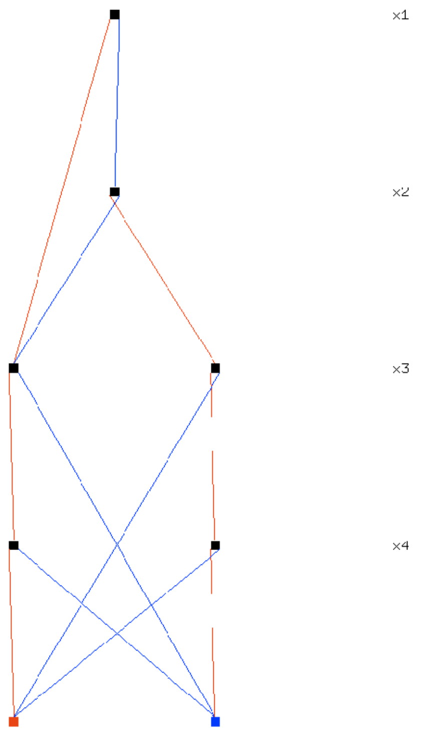


Figure 41. $x_1x_2 \oplus x_3x_4 \oplus x_1 \oplus x_3 \oplus x_4$

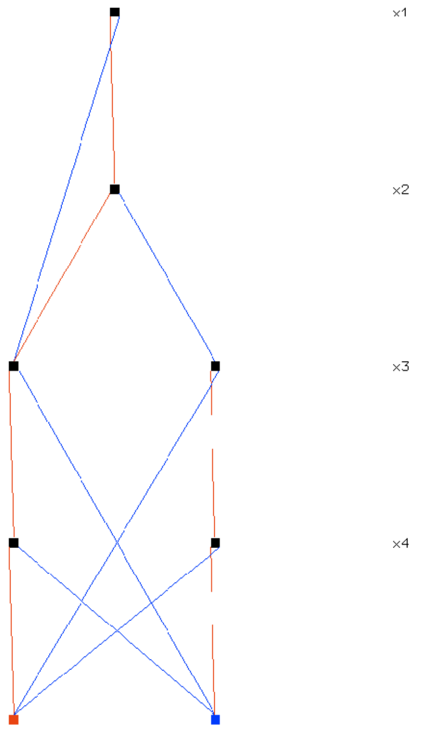


Figure 42. $x_1x_2 \oplus x_3x_4 \oplus x_2 \oplus x_3 \oplus x_4$

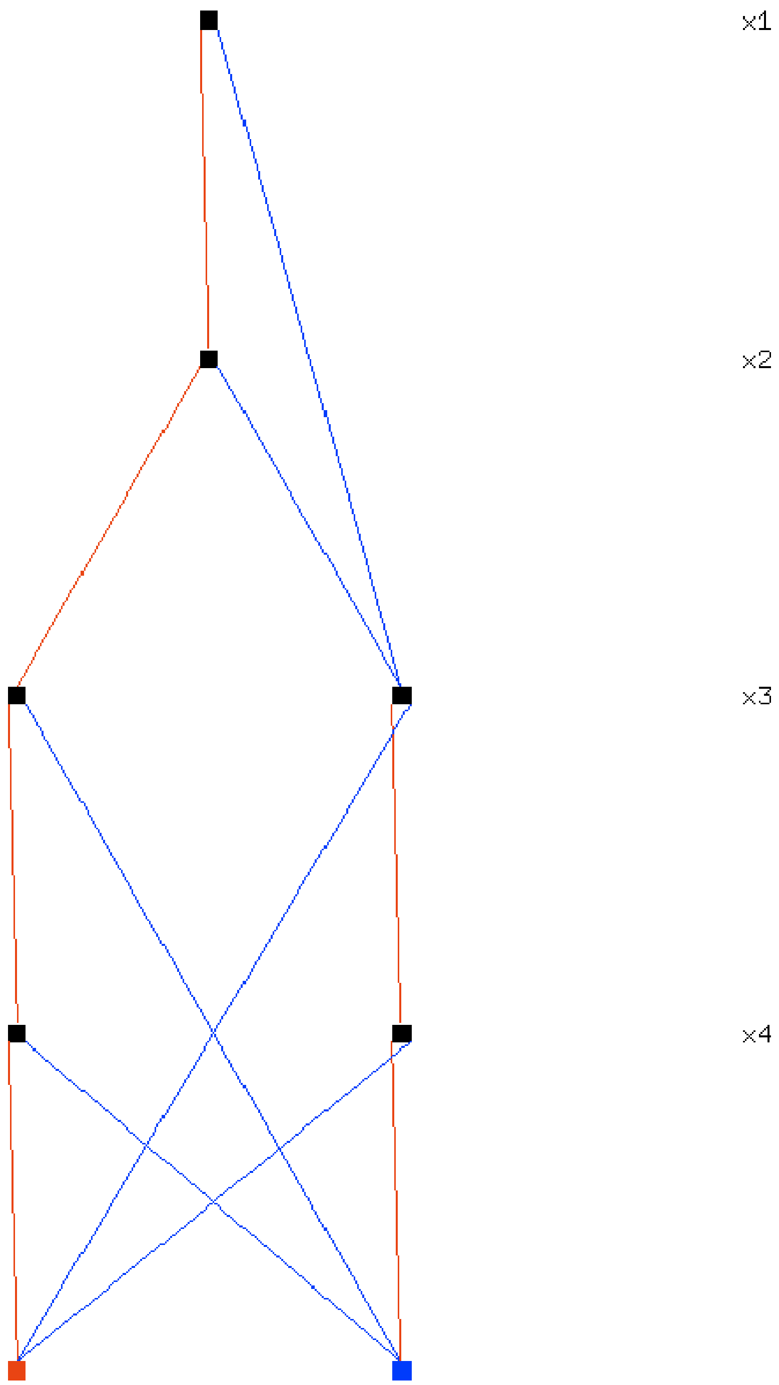


Figure 43. $x_1x_2 \oplus x_3x_4 \oplus x_1 \oplus x_2 \oplus x_3 \oplus x_4$

C. PARTIAL DISJOINT AFFINE CLASS OF ORDER 6

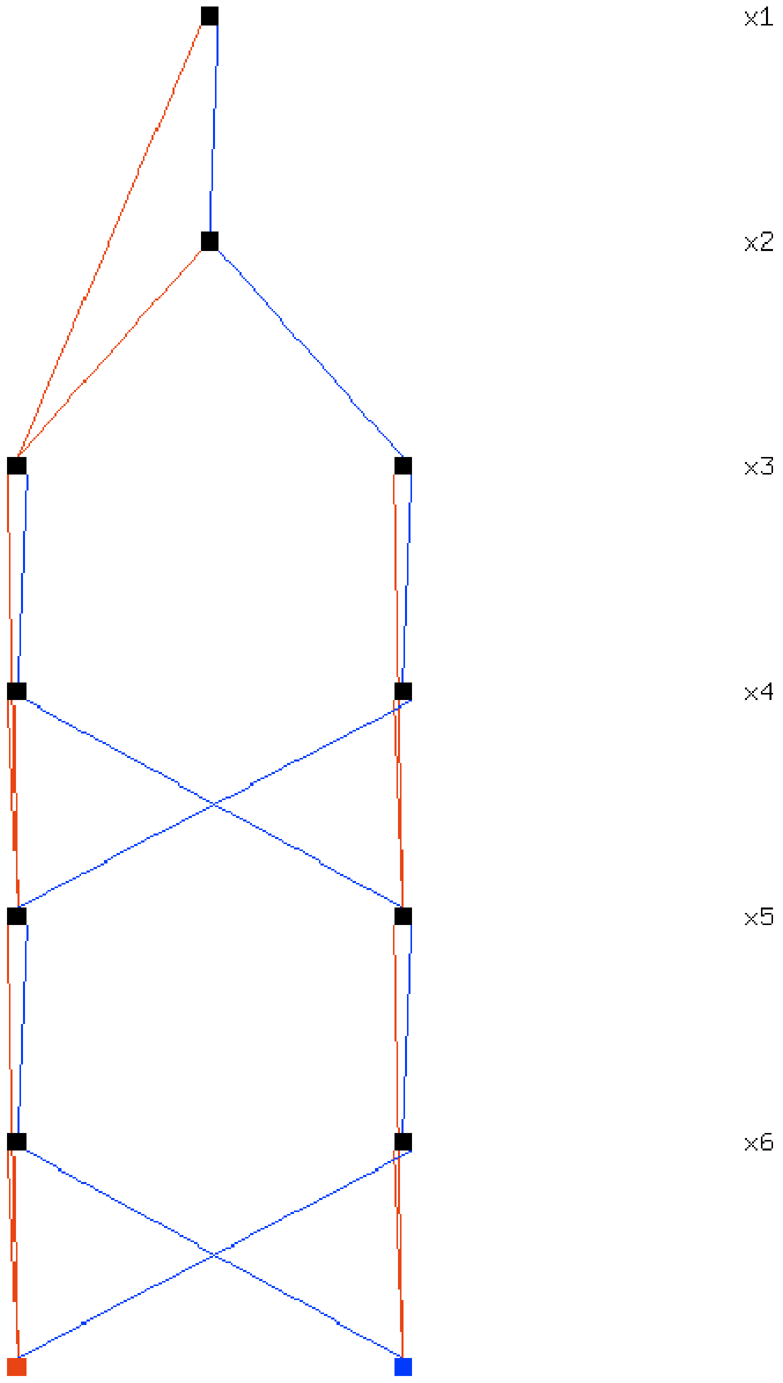


Figure 44. DQF of Order 6 ($x_1x_2 \oplus x_3x_4 \oplus x_5x_6$).

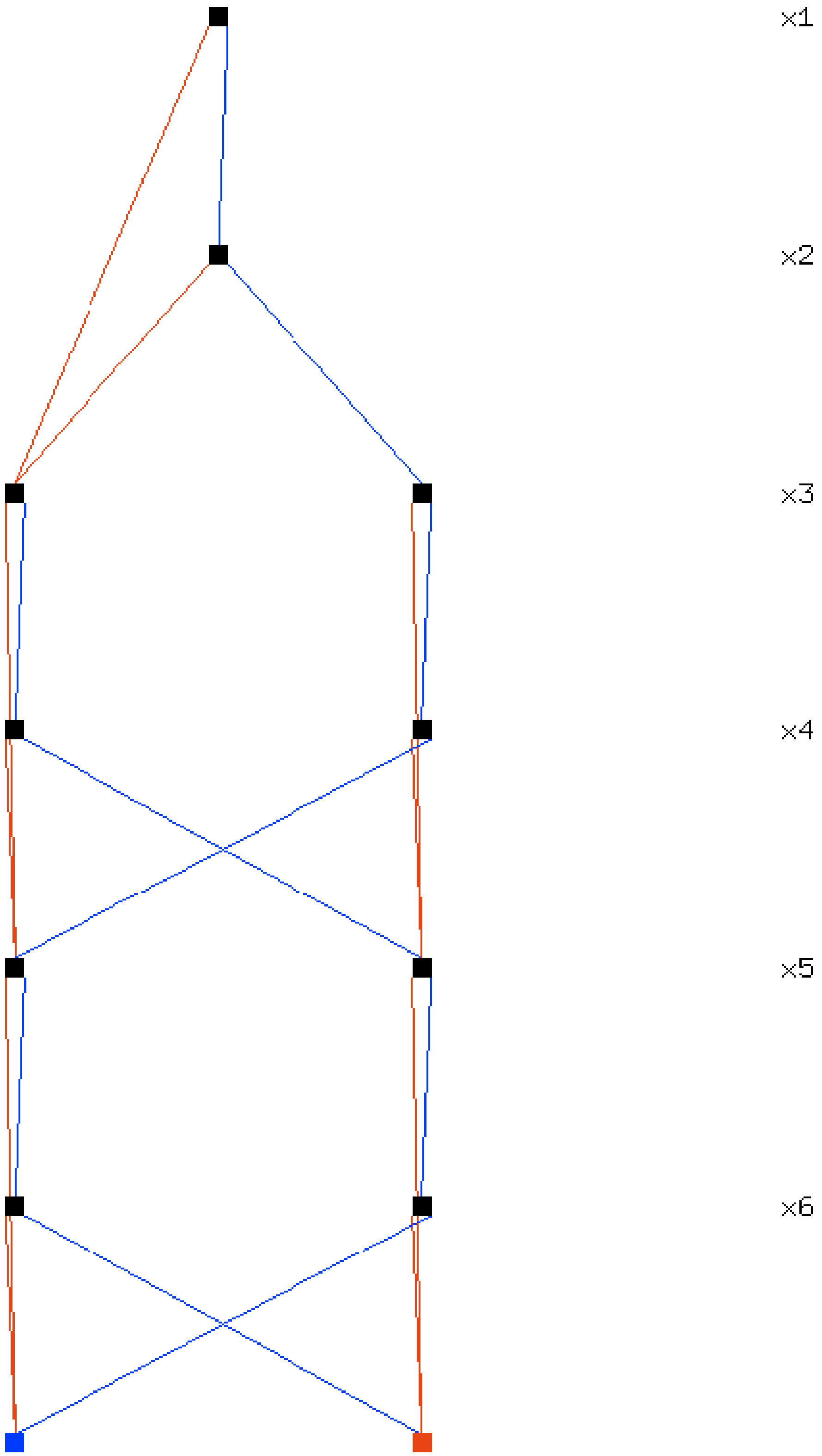


Figure 45. Complement of DQF ($x_1x_2 \oplus x_3x_4 \oplus x_5x_6 \oplus 1$).

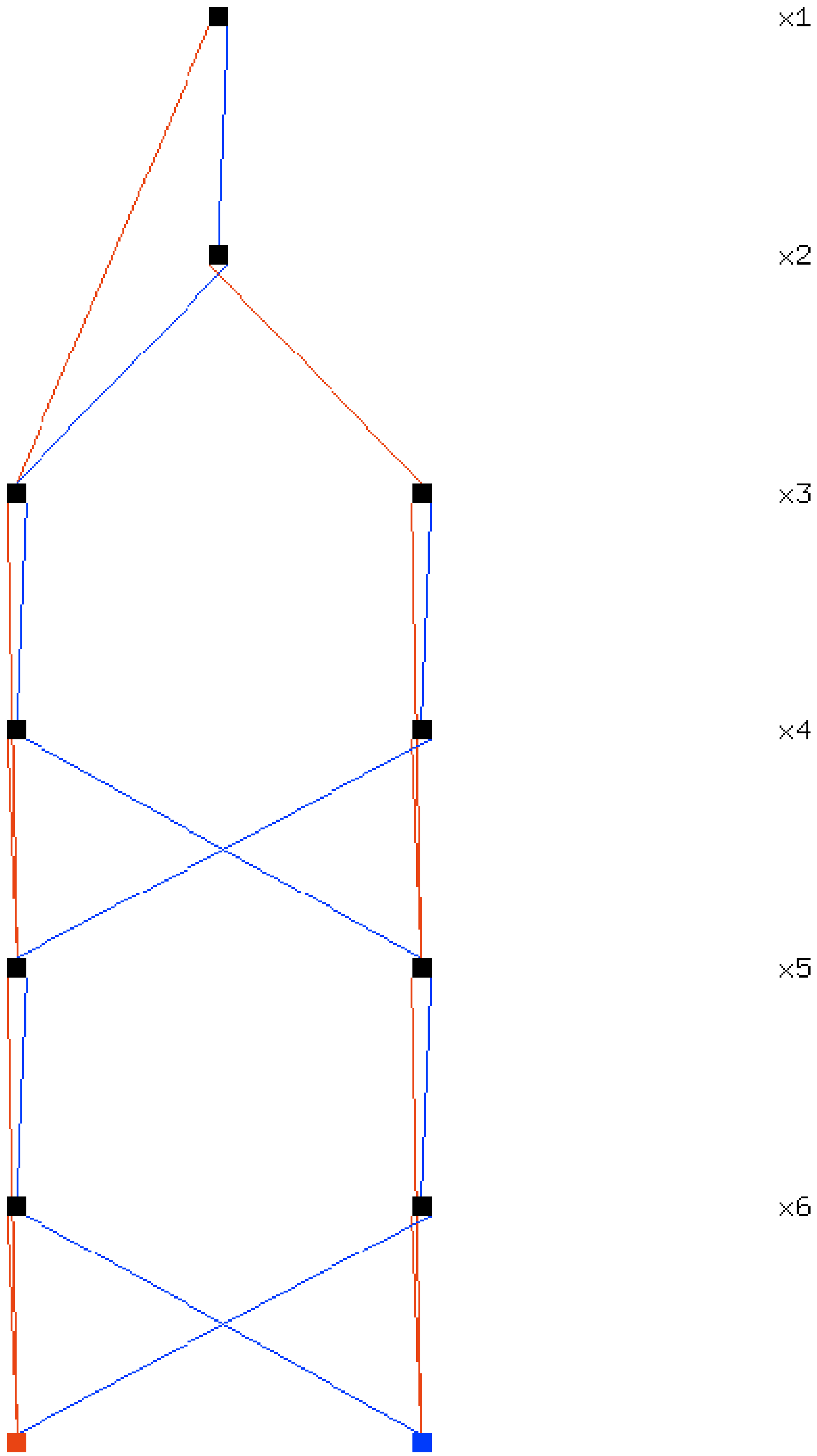


Figure 46. $x_1x_2 \oplus x_3x_4 \oplus x_5x_6 \oplus x_1$

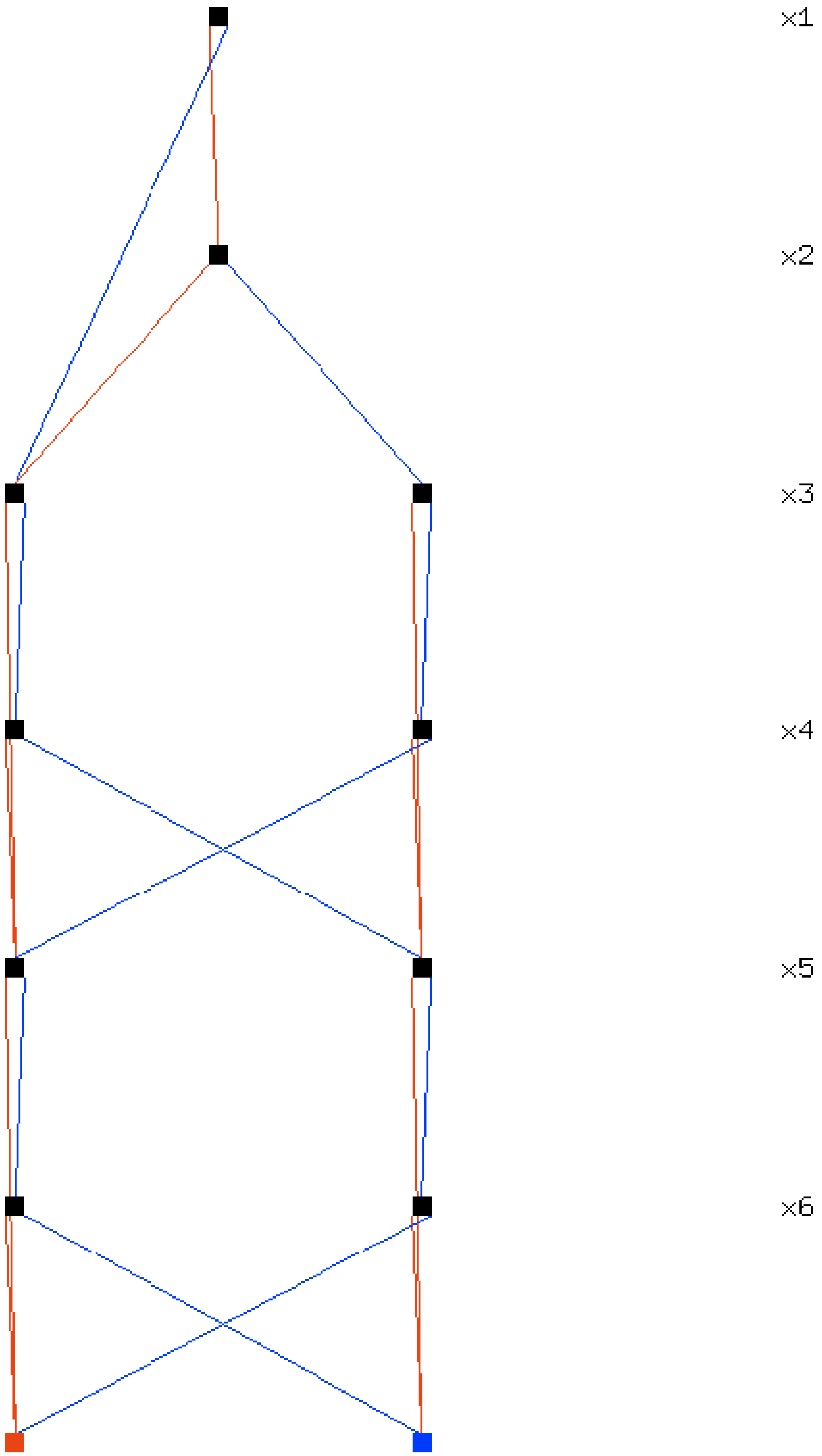


Figure 47. $x_1x_2 \oplus x_3x_4 \oplus x_5x_6 \oplus x_2$

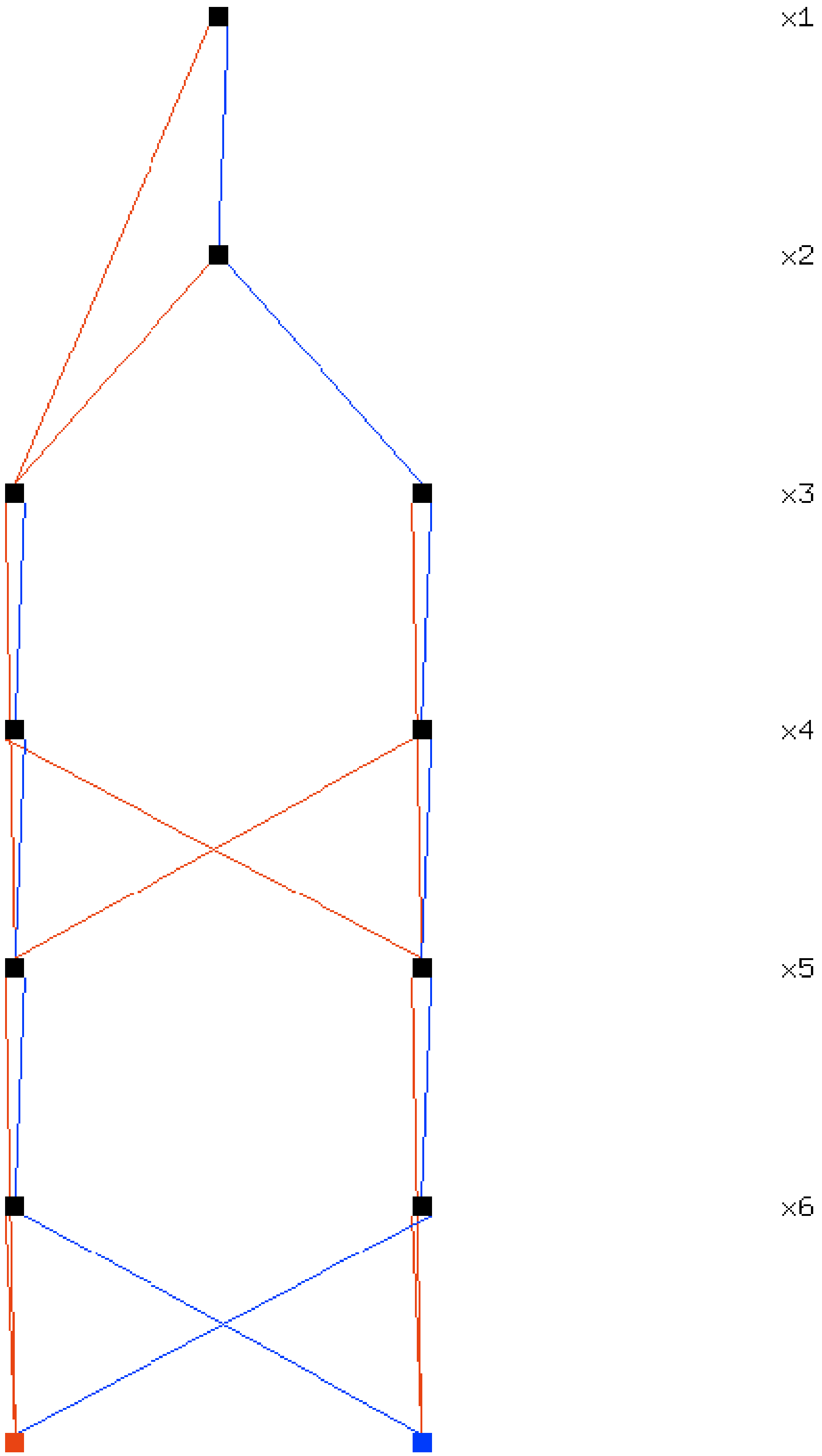


Figure 48. $x_1x_2 \oplus x_3x_4 \oplus x_5x_6 \oplus x_3$

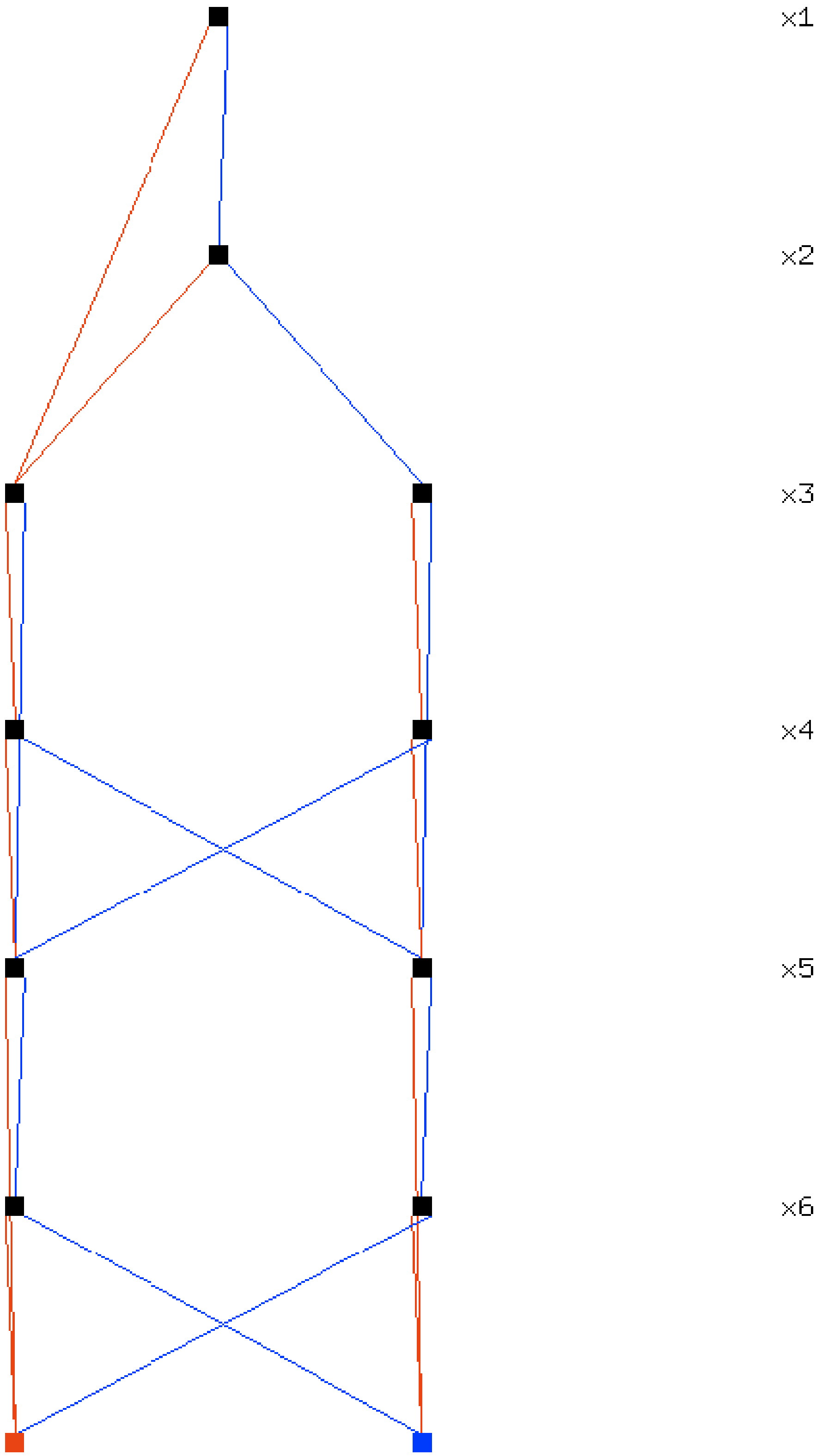


Figure 49. $x_1x_2 \oplus x_3x_4 \oplus x_5x_6 \oplus x_4$

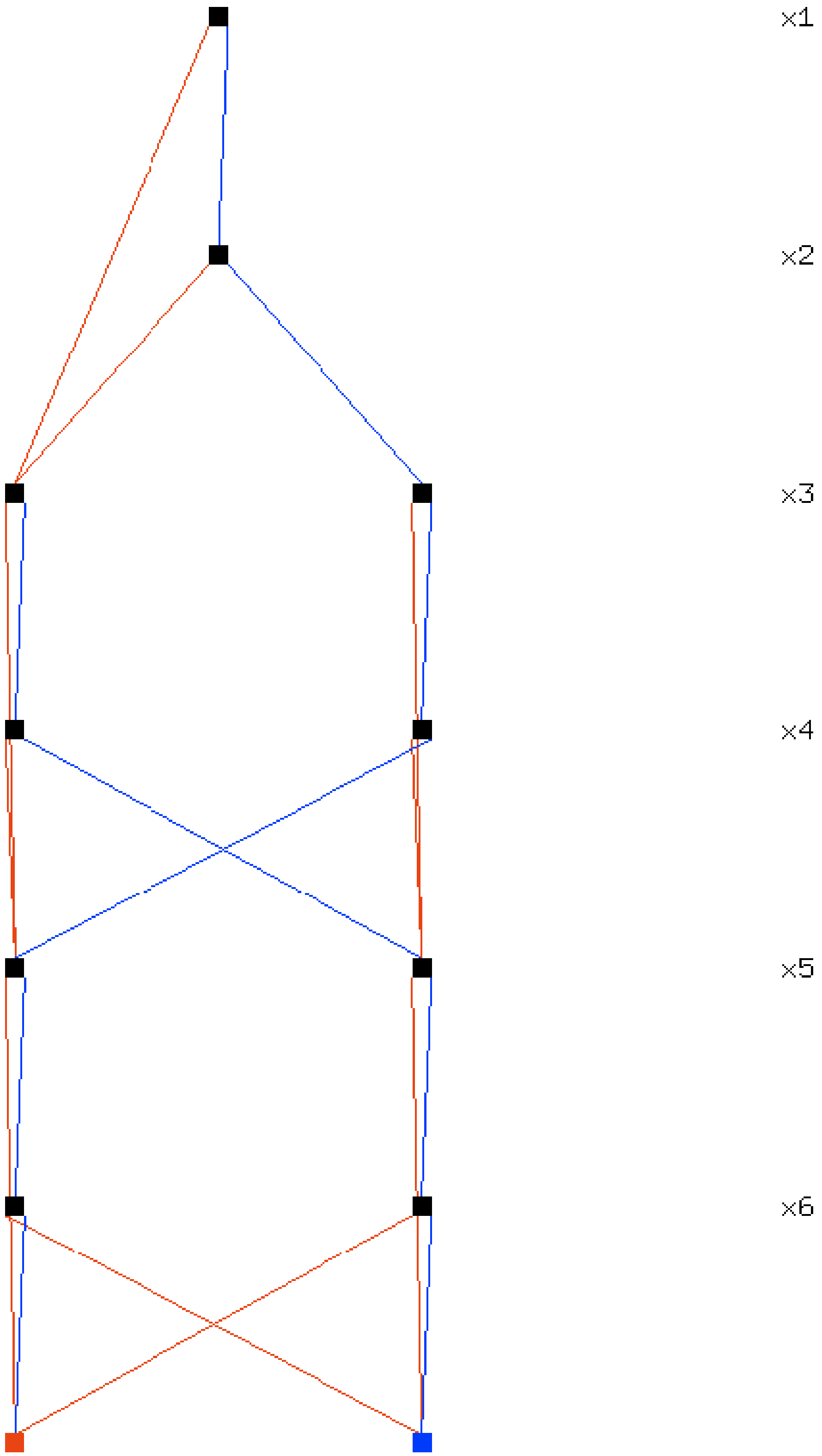


Figure 50. $x_1x_2 \oplus x_3x_4 \oplus x_5x_6 \oplus x_5$

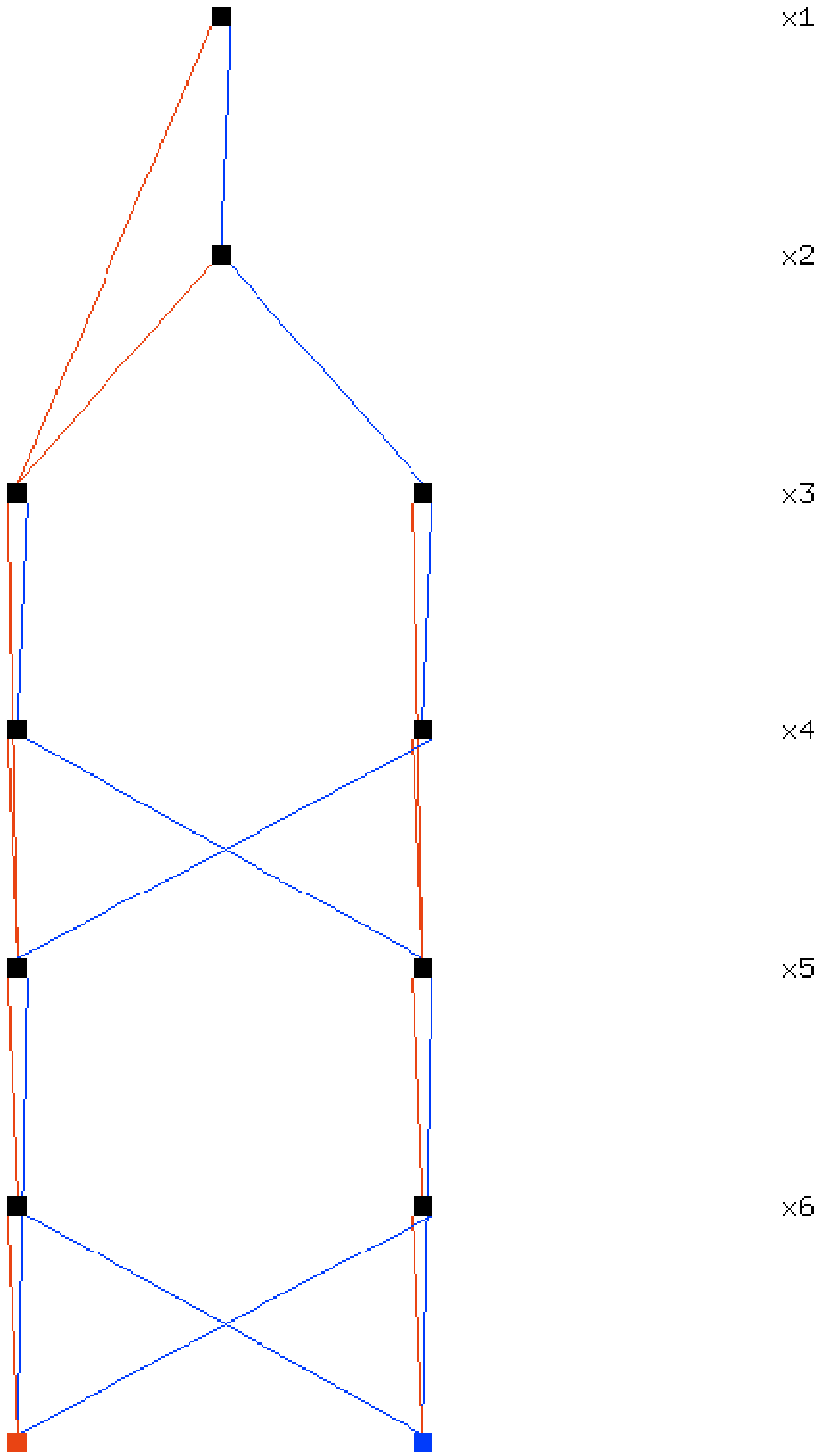


Figure 51. $x_1x_2 \oplus x_3x_4 \oplus x_5x_6 \oplus x_6$

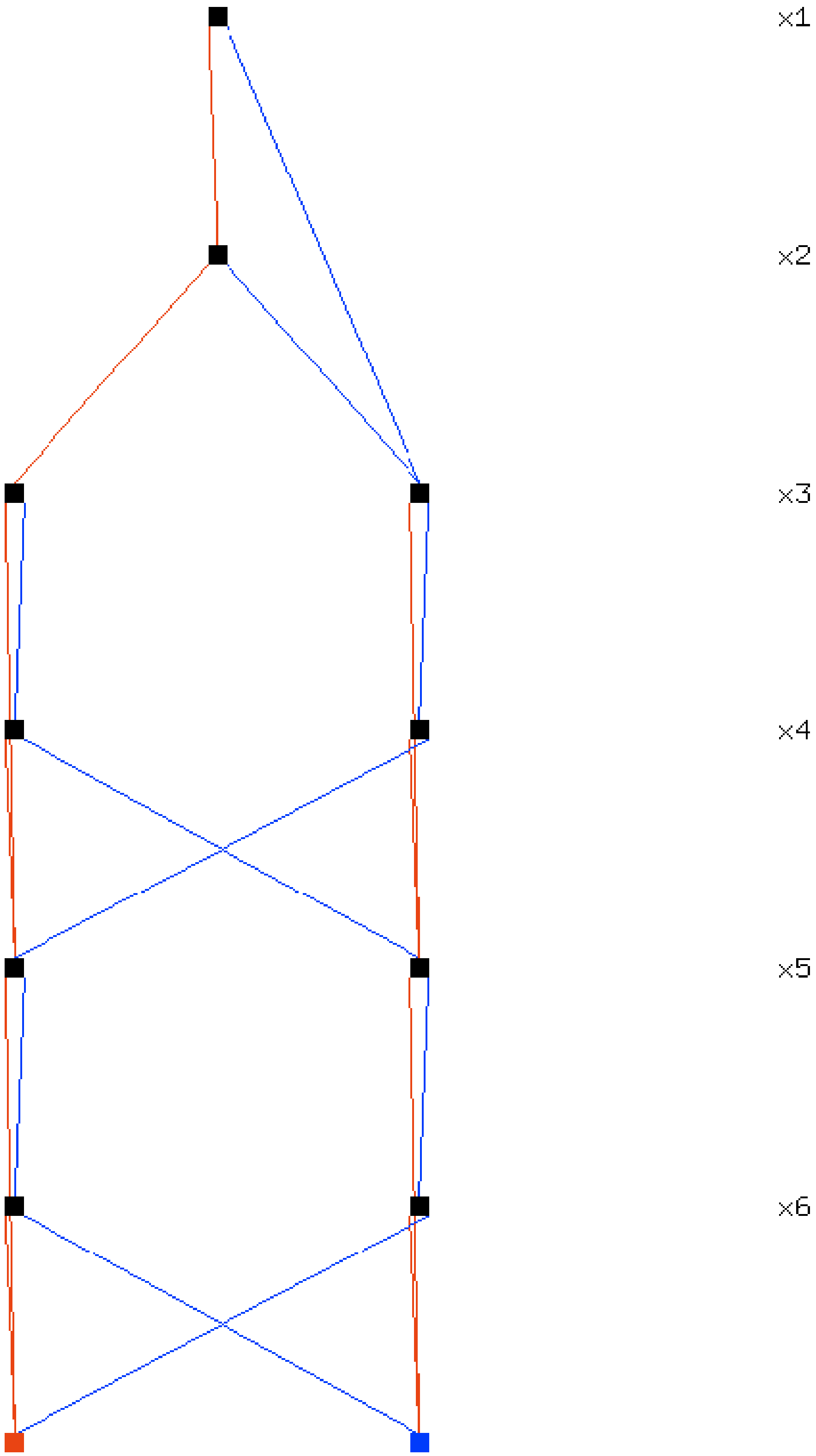


Figure 52. $x_1x_2 \oplus x_3x_4 \oplus x_5x_6 \oplus x_1 \oplus x_2$

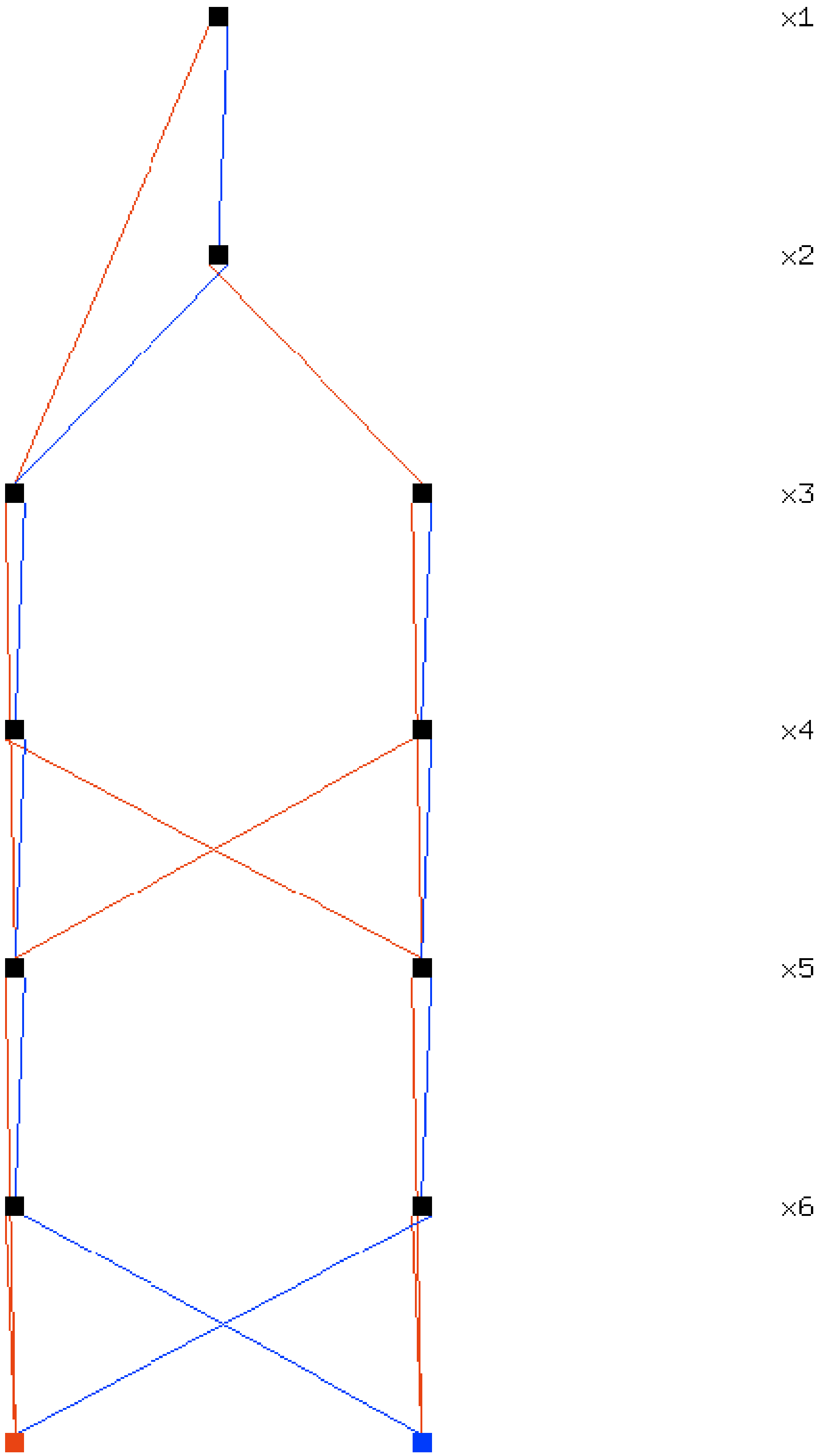


Figure 53. $x_1x_2 \oplus x_3x_4 \oplus x_5x_6 \oplus x_1 \oplus x_3$

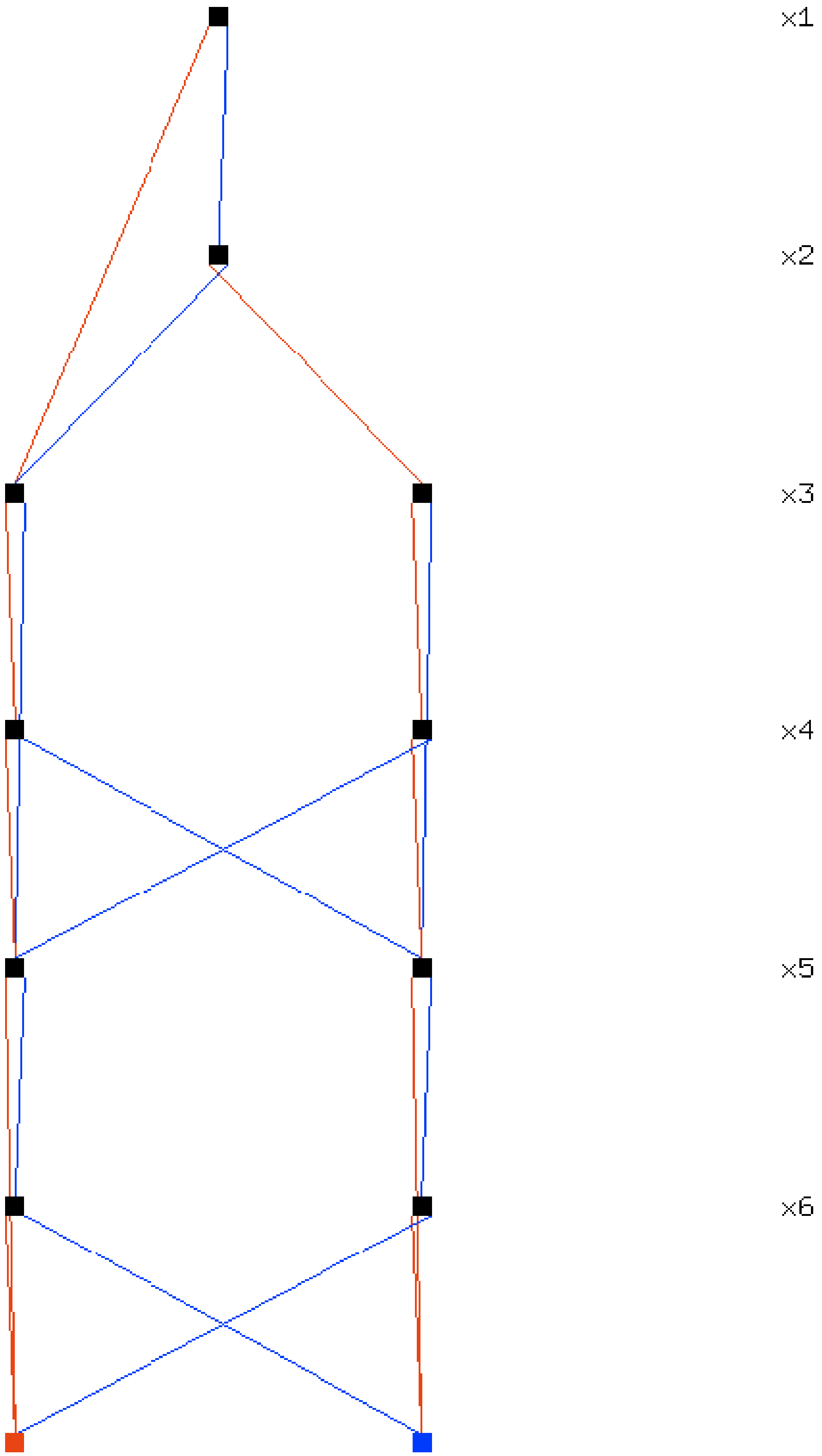


Figure 54. $x_1x_2 \oplus x_3x_4 \oplus x_5x_6 \oplus x_1 \oplus x_4$

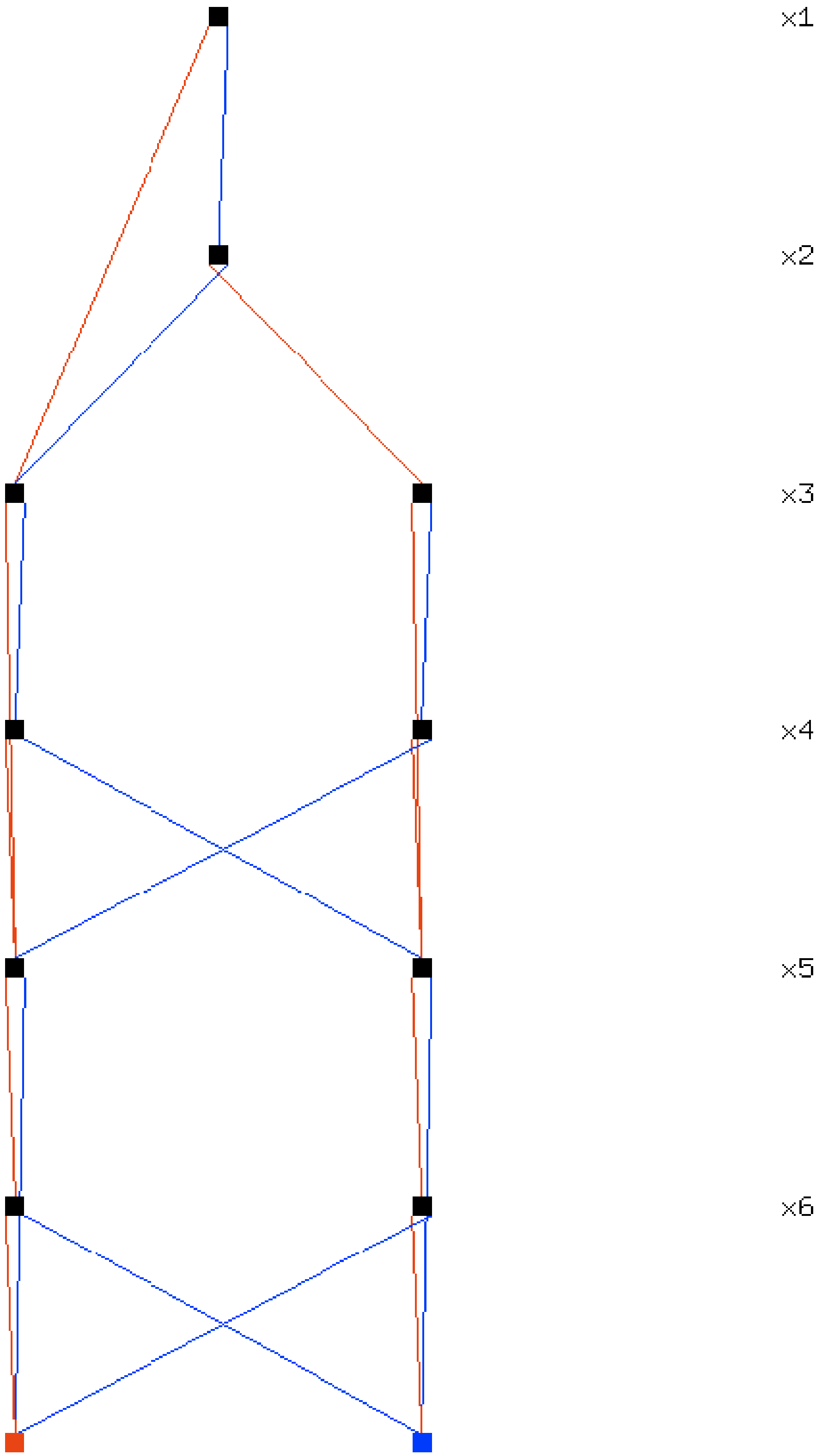


Figure 55. $x_1x_2 \oplus x_3x_4 \oplus x_5x_6 \oplus x_1 \oplus x_6$

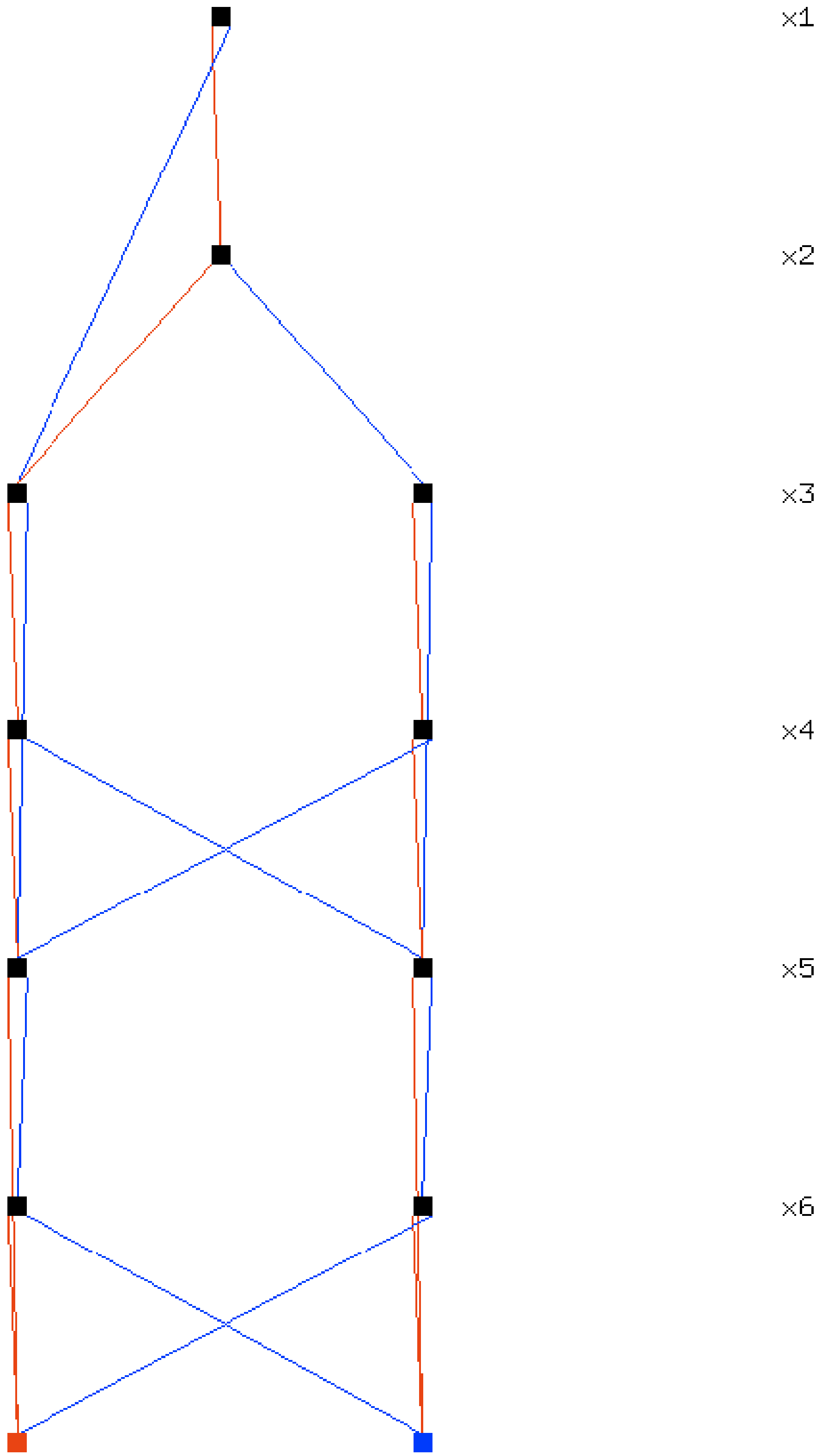


Figure 56. $x_1x_2 \oplus x_3x_4 \oplus x_5x_6 \oplus x_2 \oplus x_4$

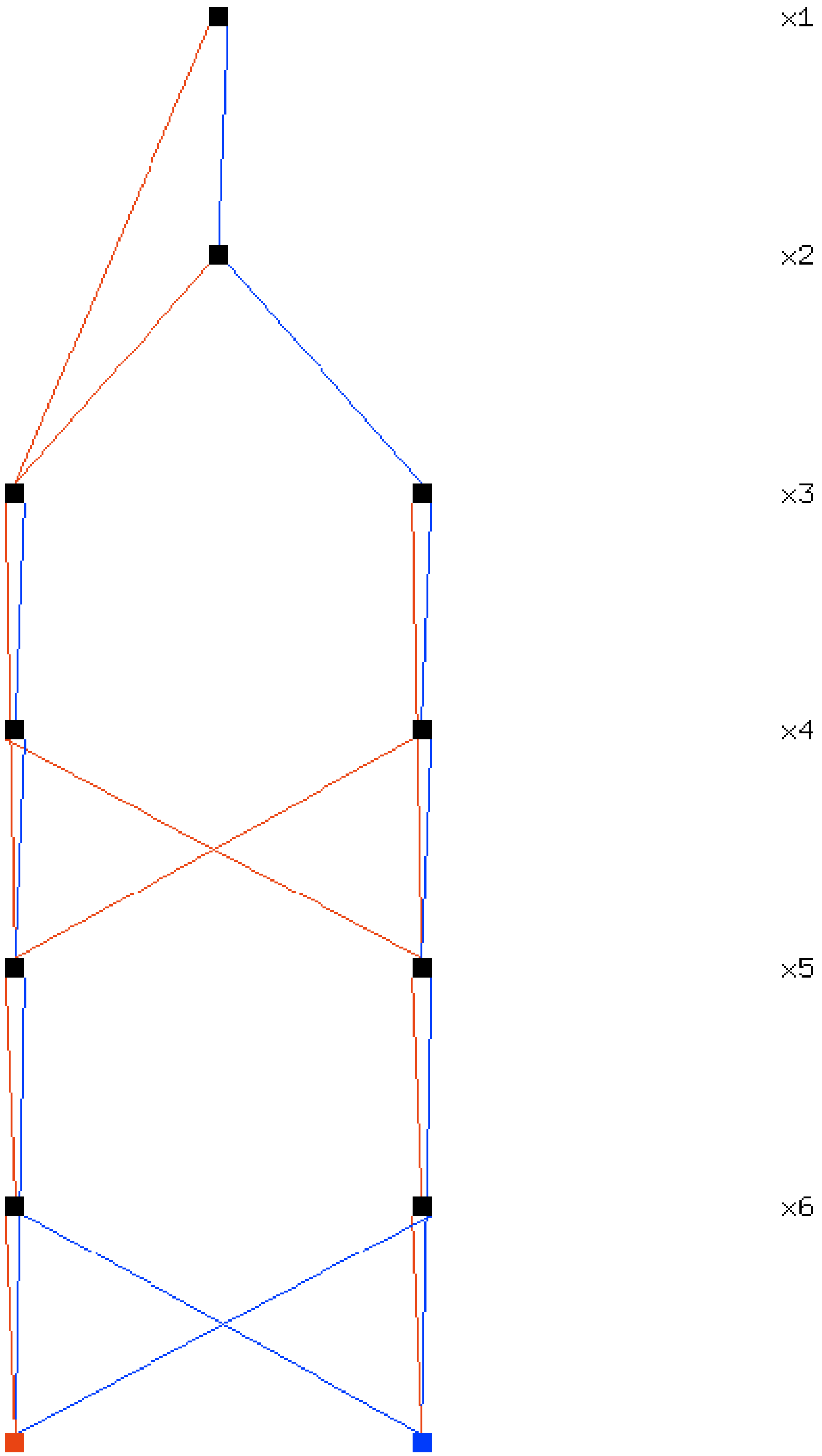


Figure 57. $x_1x_2 \oplus x_3x_4 \oplus x_5x_6 \oplus x_3 \oplus x_6$

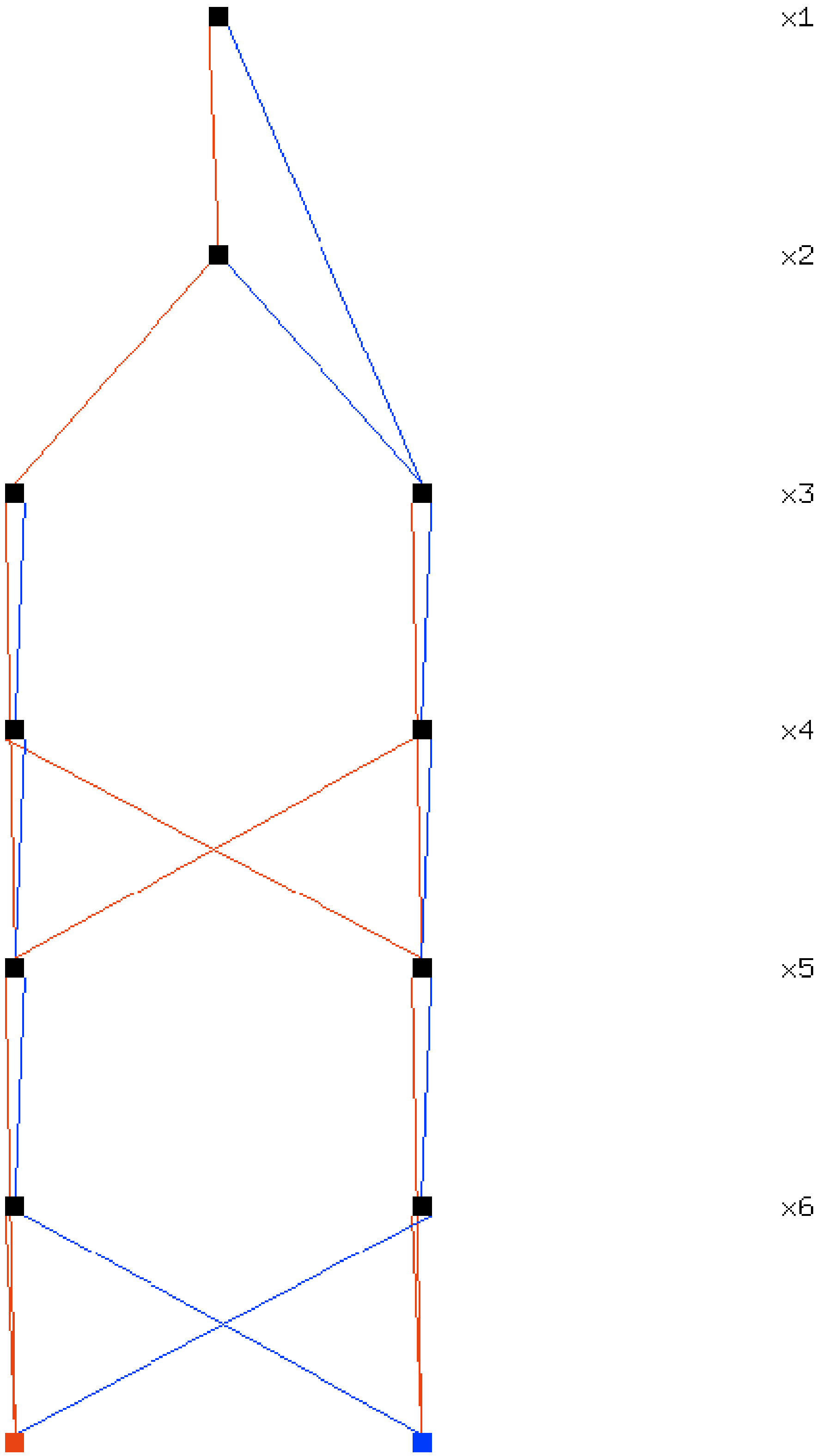


Figure 58. $x_1x_2 \oplus x_3x_4 \oplus x_5x_6 \oplus x_1 \oplus x_2 \oplus x_3$

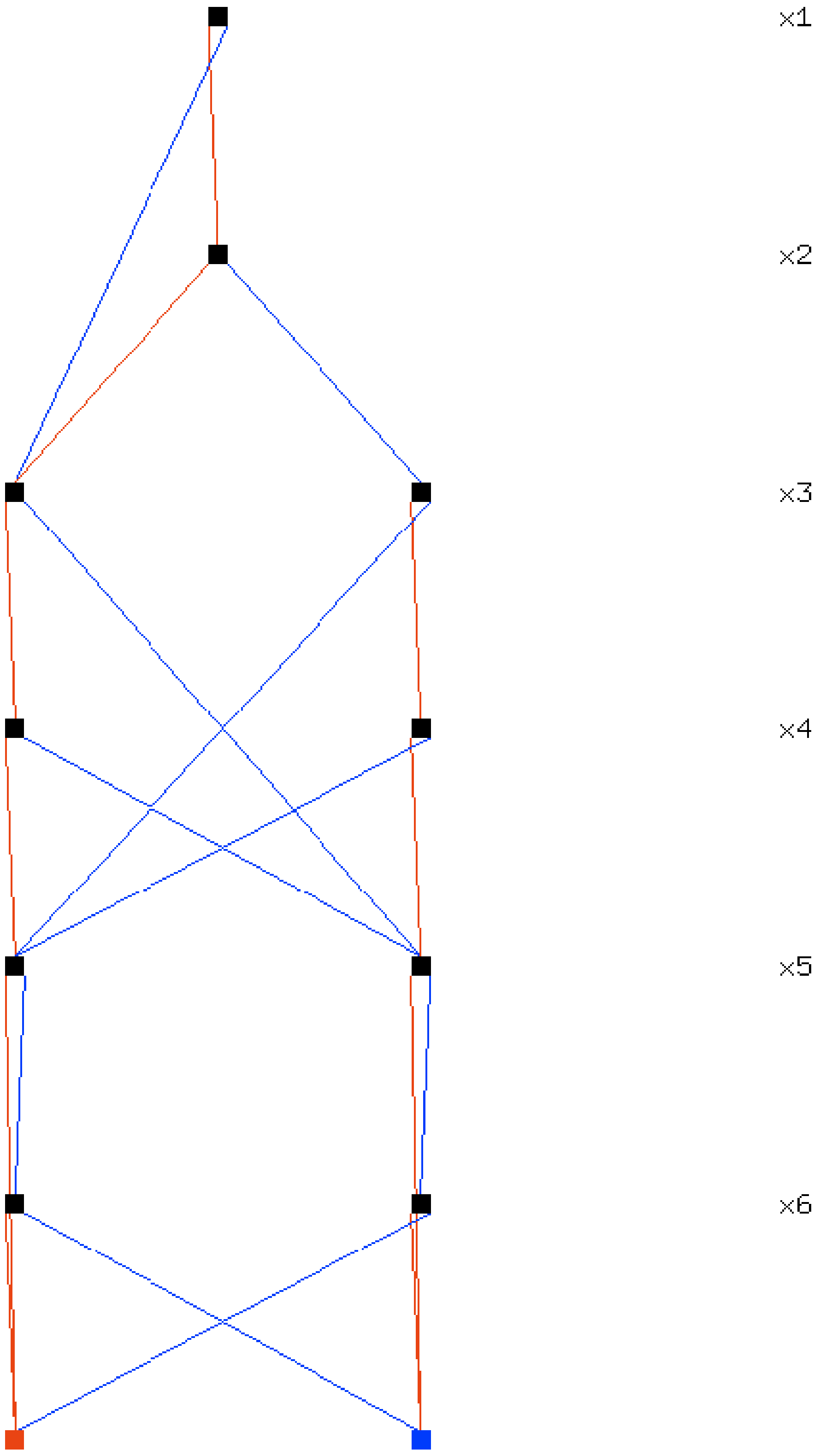


Figure 59. $x_1x_2 \oplus x_3x_4 \oplus x_5x_6 \oplus x_2 \oplus x_3 \oplus x_4$

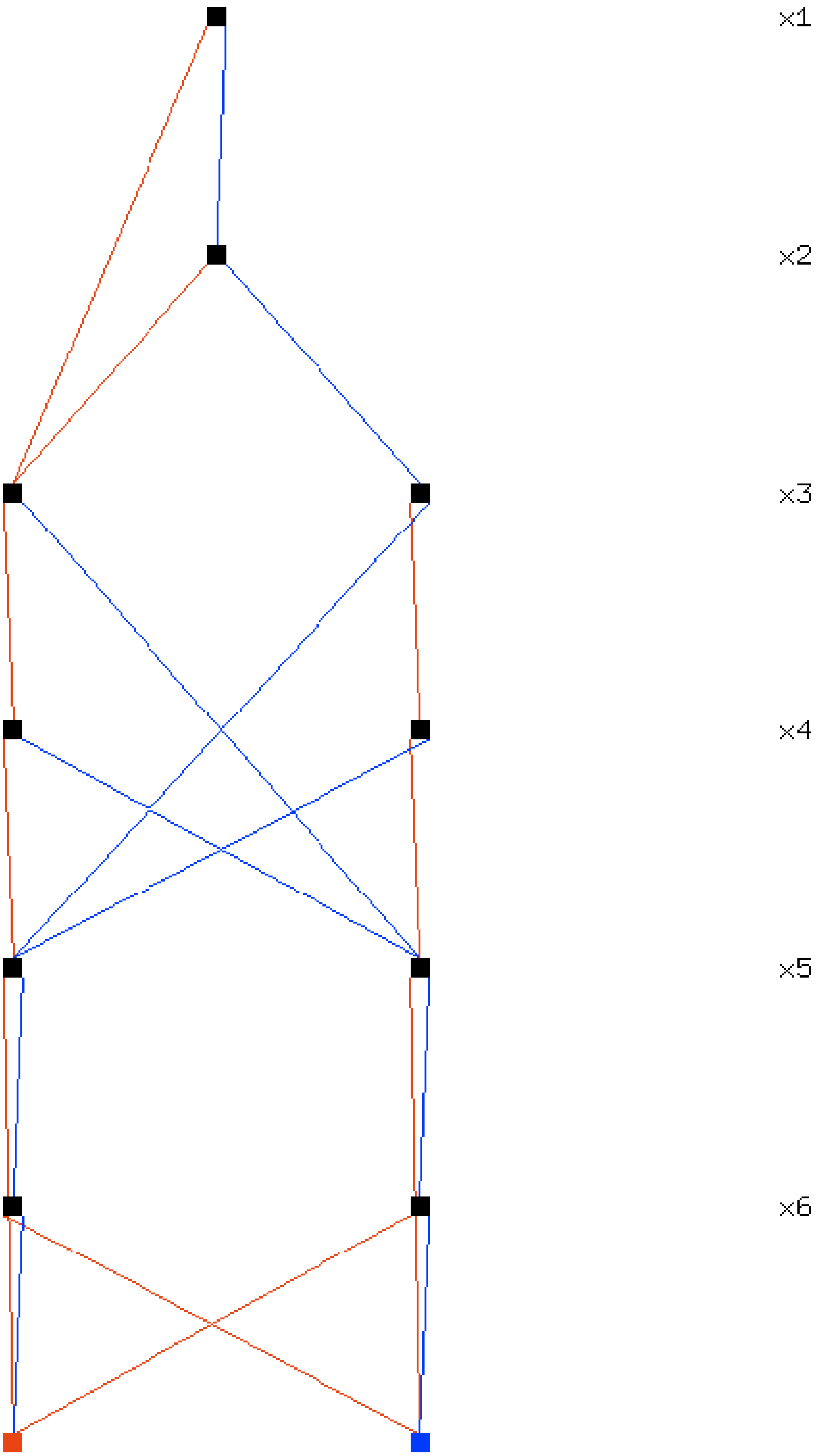


Figure 60. $x_1x_2 \oplus x_3x_4 \oplus x_5x_6 \oplus x_3 \oplus x_4 \oplus x_5$

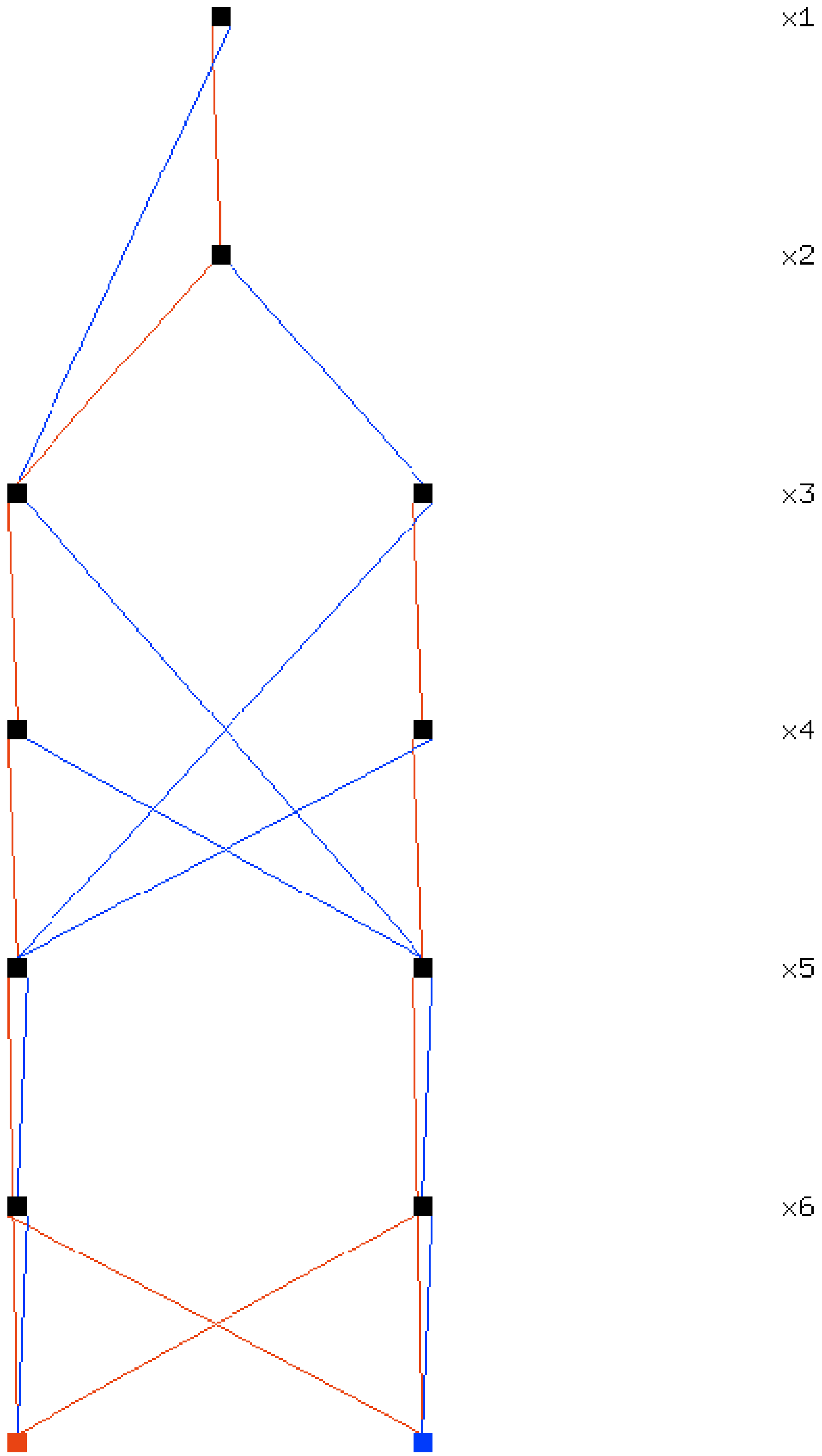


Figure 61. $x_1x_2 \oplus x_3x_4 \oplus x_5x_6 \oplus x_2 \oplus x_3 \oplus x_4 \oplus x_5$

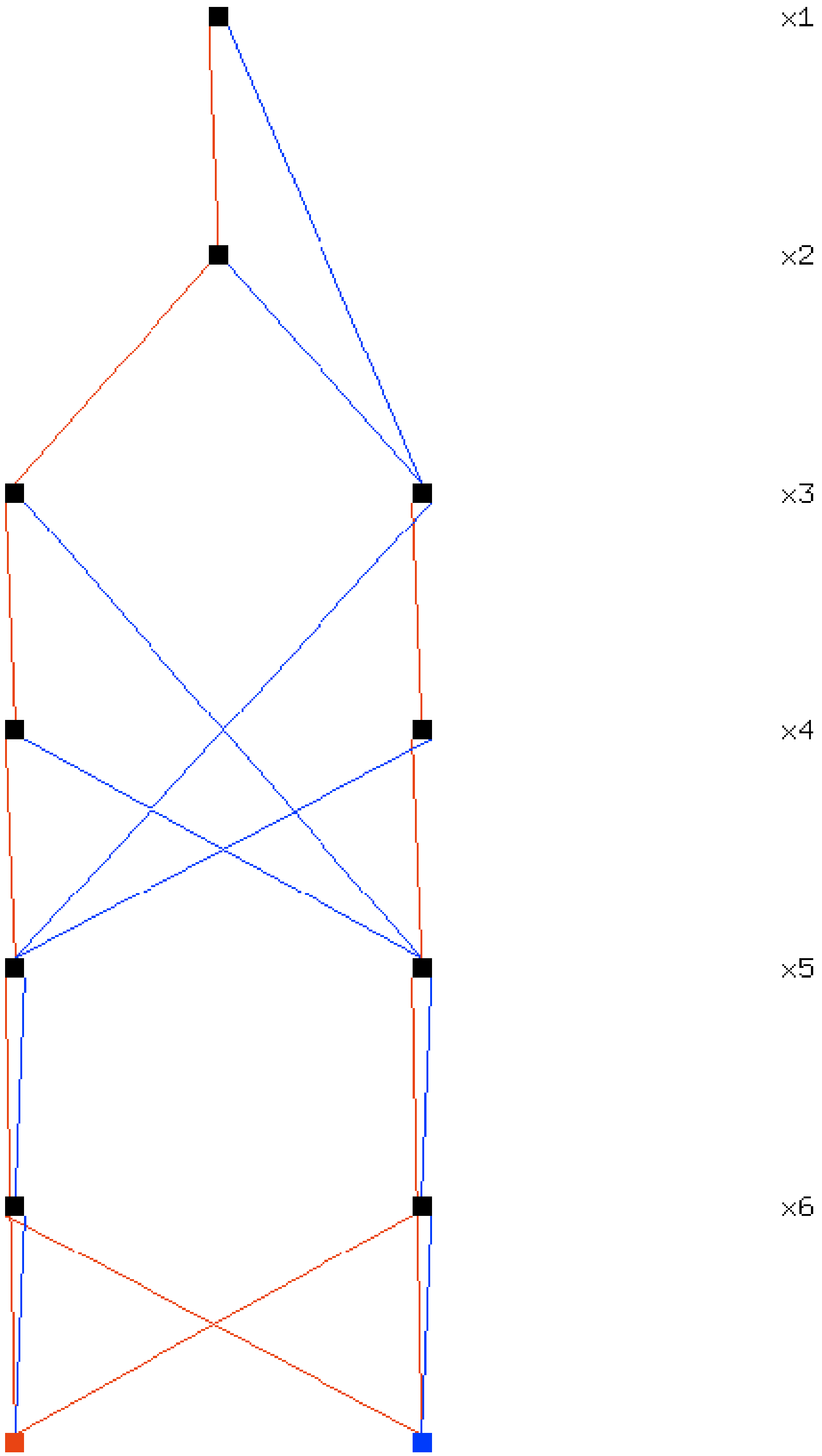


Figure 62. $x_1x_2 \oplus x_3x_4 \oplus x_5x_6 \oplus x_1 \oplus x_2 \oplus x_3 \oplus x_4 \oplus x_5$

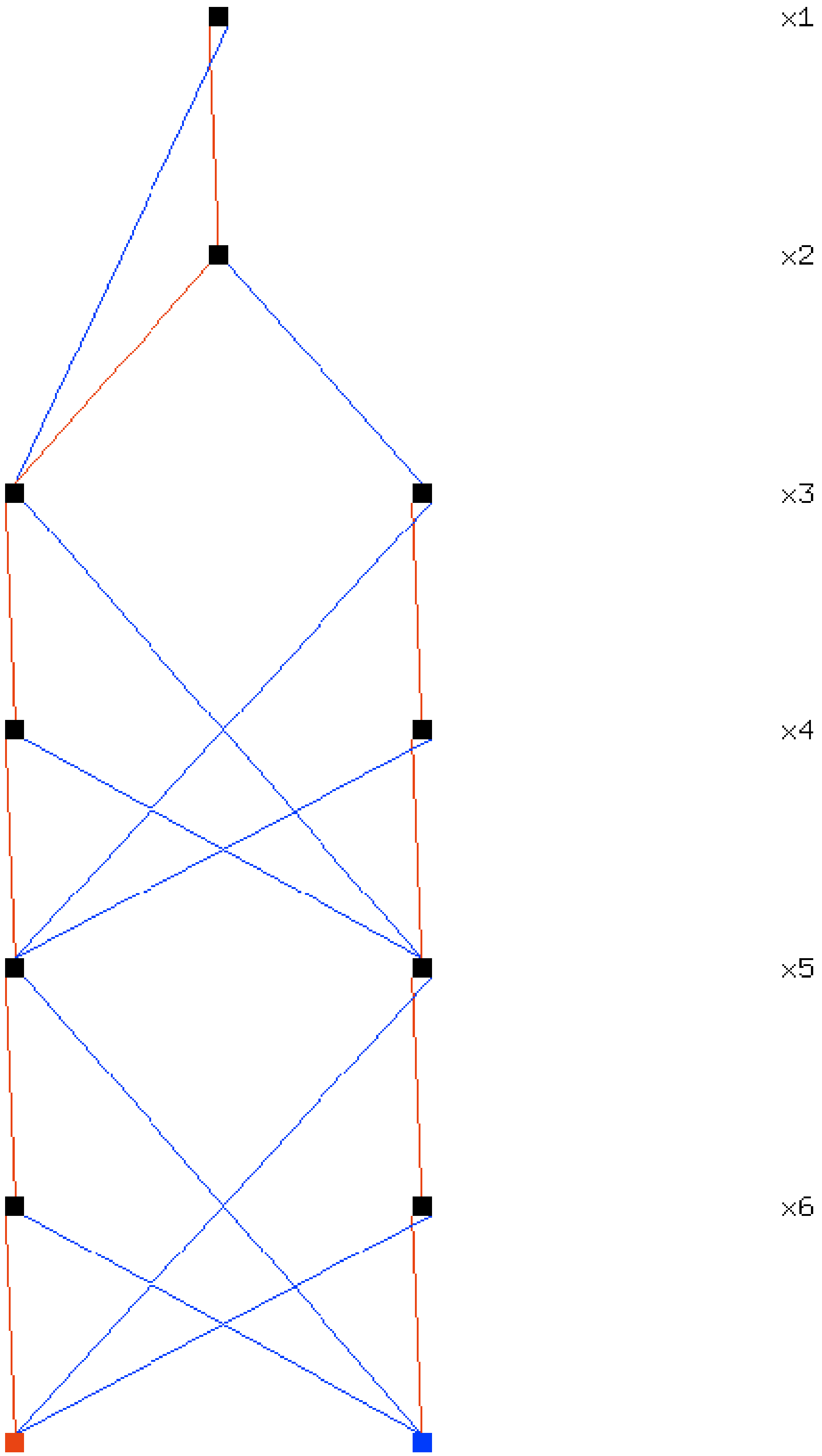


Figure 63. $x_1x_2 \oplus x_3x_4 \oplus x_5x_6 \oplus x_2 \oplus x_3 \oplus x_4 \oplus x_5 \oplus x_6$

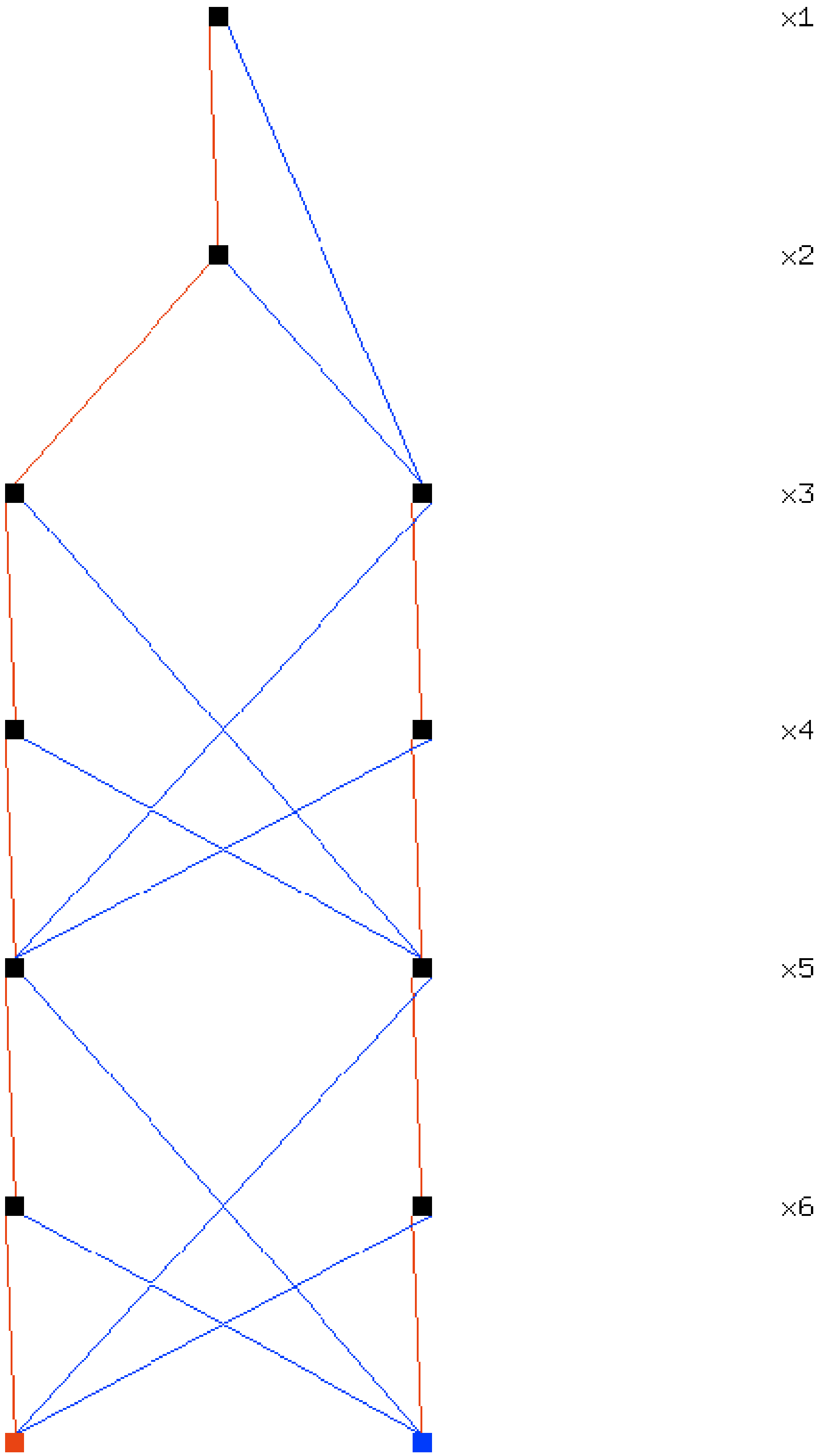


Figure 64. $x_1x_2 \oplus x_3x_4 \oplus x_5x_6 \oplus x_1 \oplus x_2 \oplus x_3 \oplus x_4 \oplus x_5 \oplus x_6$

D. PARTIAL DISJOINT QUADRATIC AFFINE CLASS OF ORDER 8

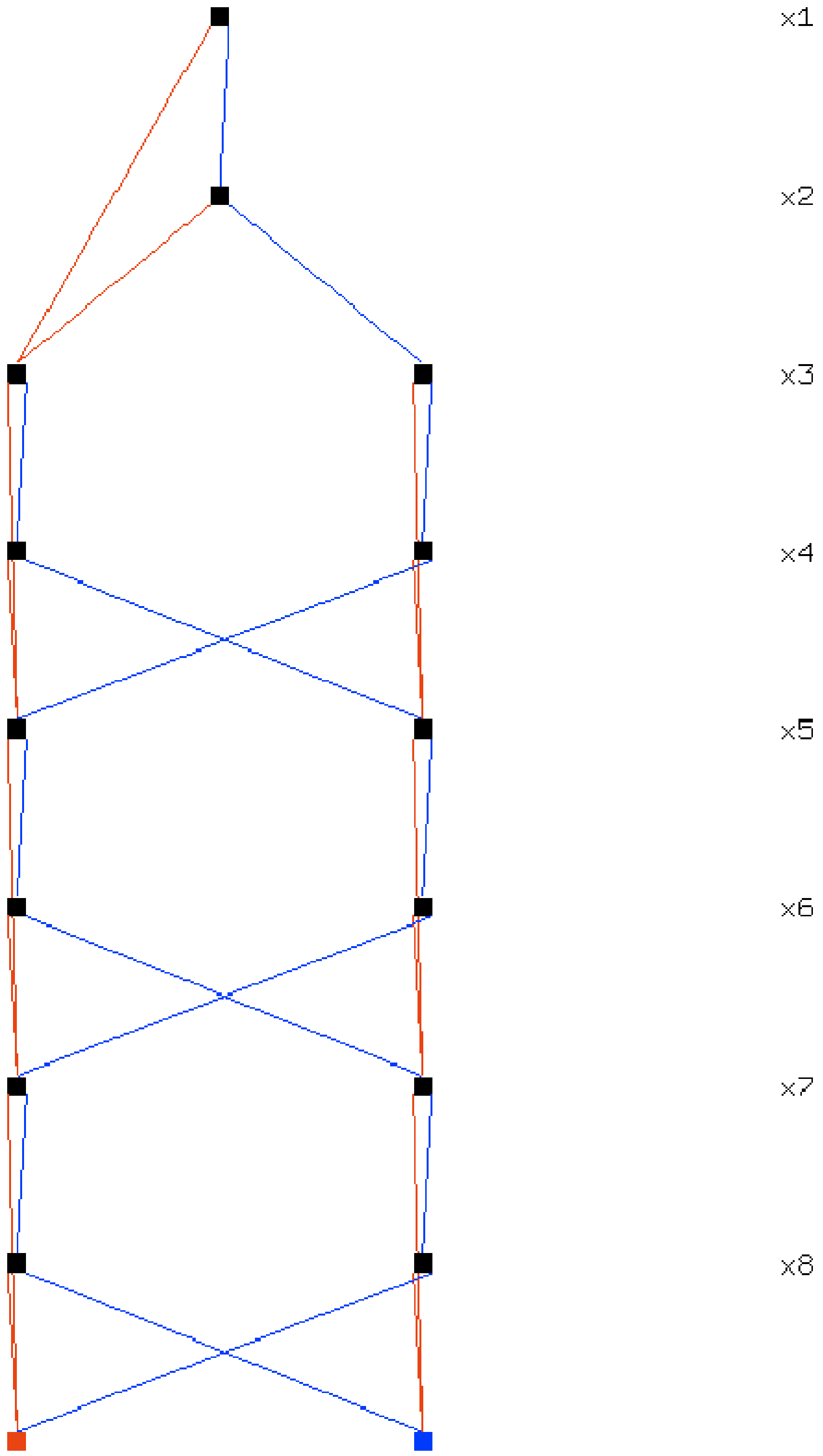


Figure 65. Order 8 DQF ($x_1x_2 \oplus x_3x_4 \oplus x_5x_6 \oplus x_7x_8$).

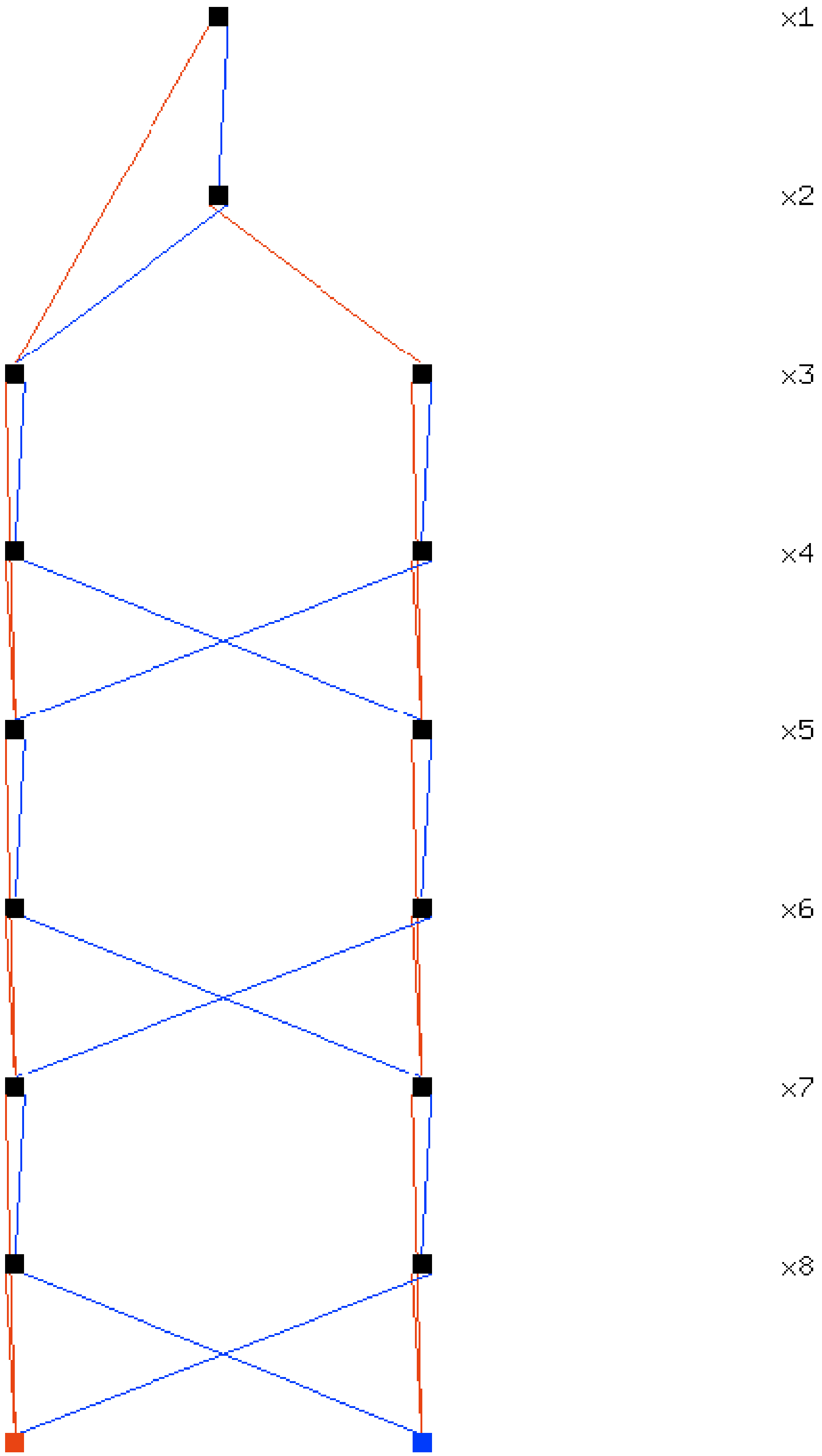


Figure 66. $x_1x_2 \oplus x_3x_4 \oplus x_5x_6 \oplus x_7x_8 \oplus x_1$

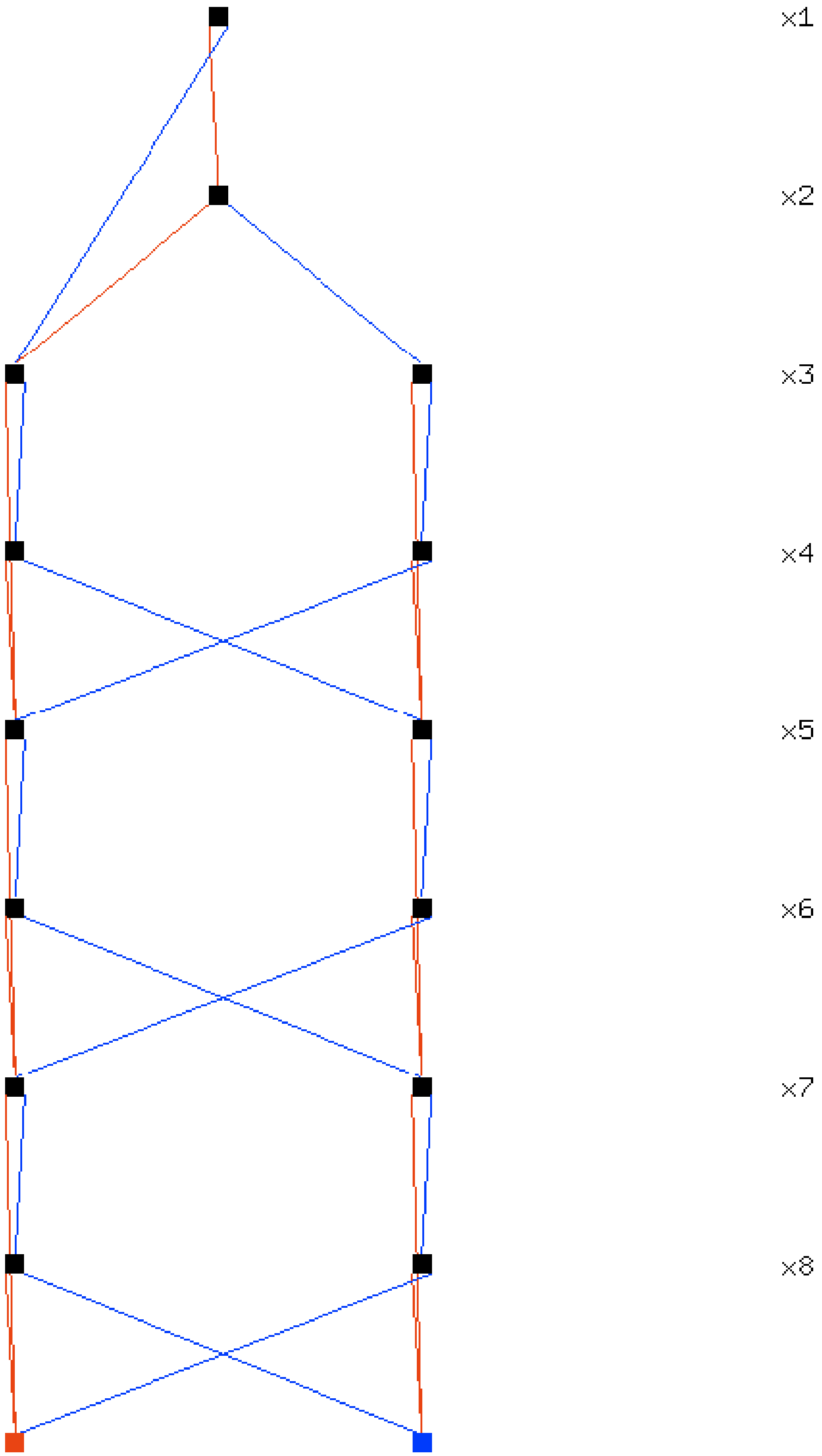


Figure 67. $x_1x_2 \oplus x_3x_4 \oplus x_5x_6 \oplus x_7x_8 \oplus x_2$

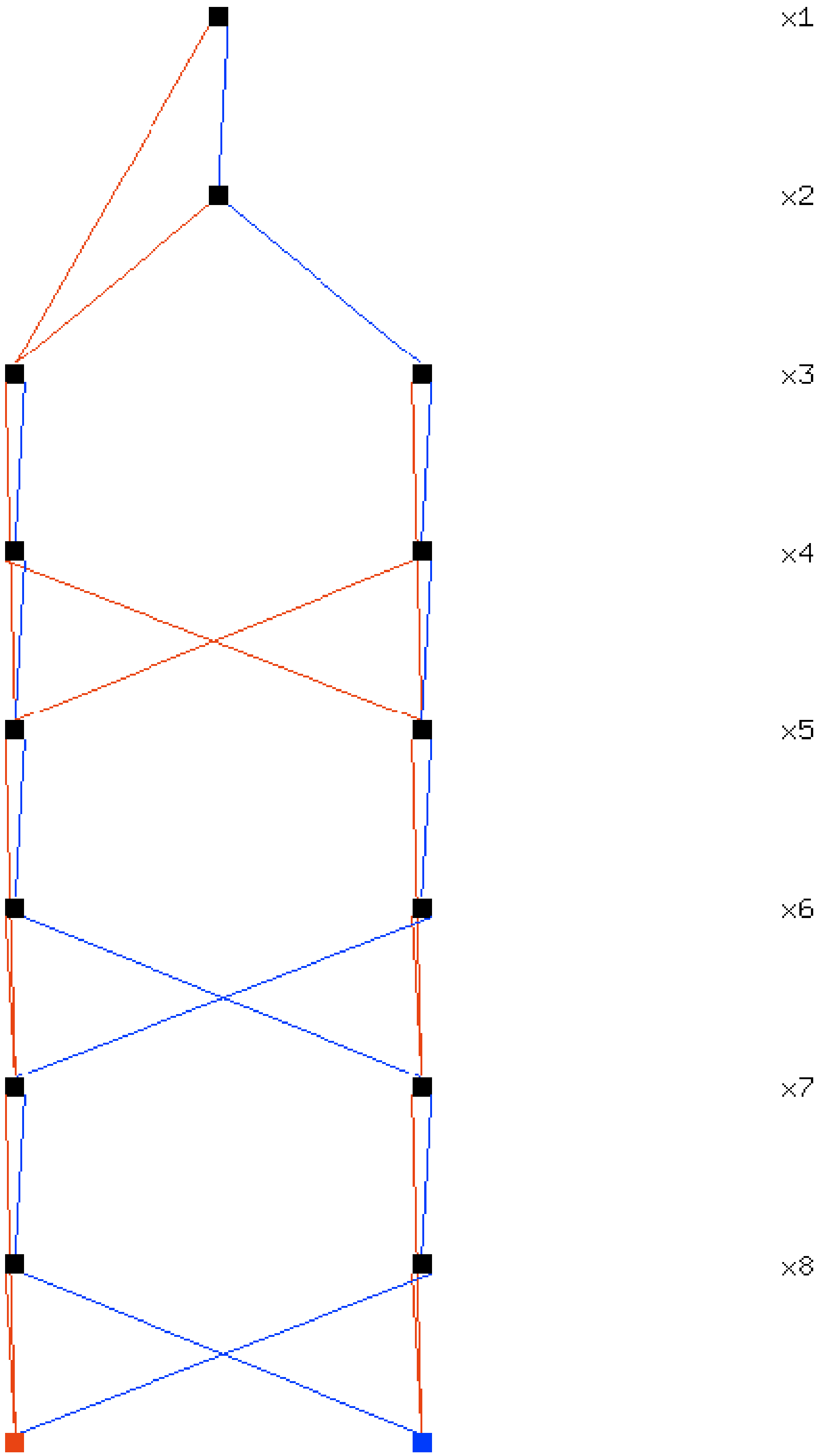


Figure 68. $x_1x_2 \oplus x_3x_4 \oplus x_5x_6 \oplus x_7x_8 \oplus x_3$

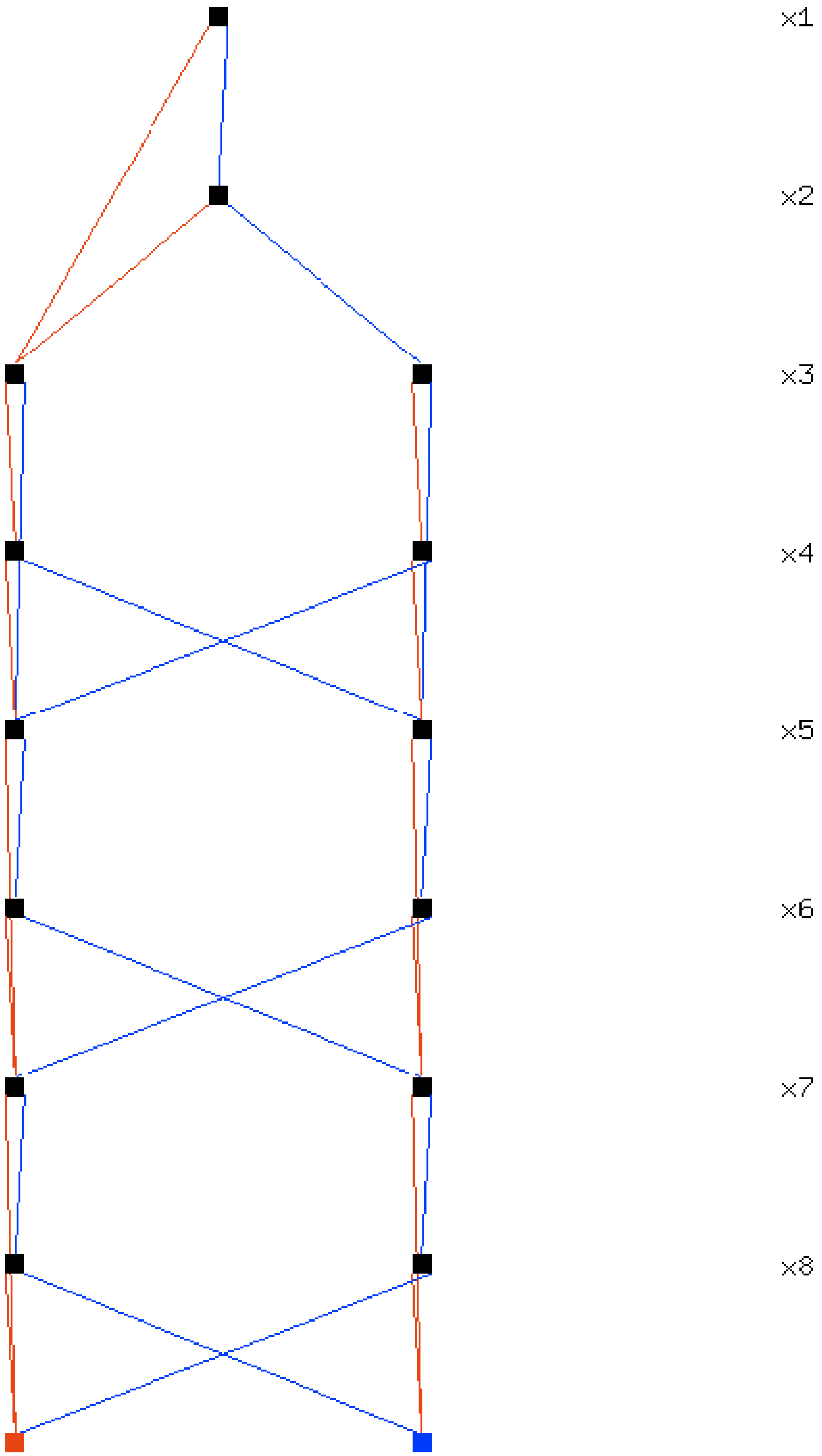


Figure 69. $x_1x_2 \oplus x_3x_4 \oplus x_5x_6 \oplus x_7x_8 \oplus x_4$

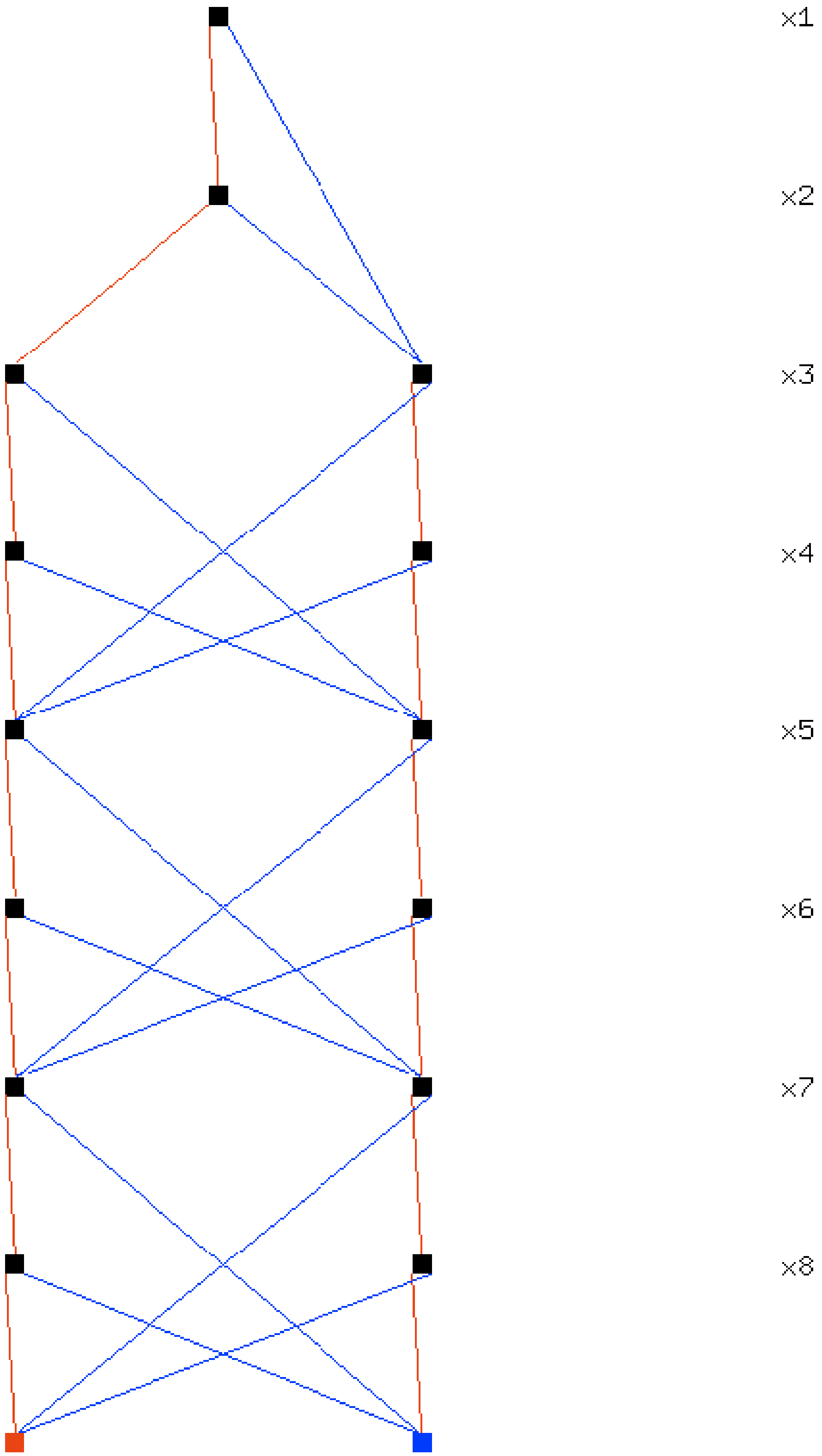


Figure 70. $x_1x_2 \oplus x_3x_4 \oplus x_5x_6 \oplus x_7x_8 \oplus x_1 \oplus x_2 \oplus x_3 \oplus x_4 \oplus x_5 \oplus x_6 \oplus x_7 \oplus x_8$

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX C: SYMMETRIC BENT FUNCTIONS

A. SYMMETRIC BENT FUNCTION OF ORDER 2

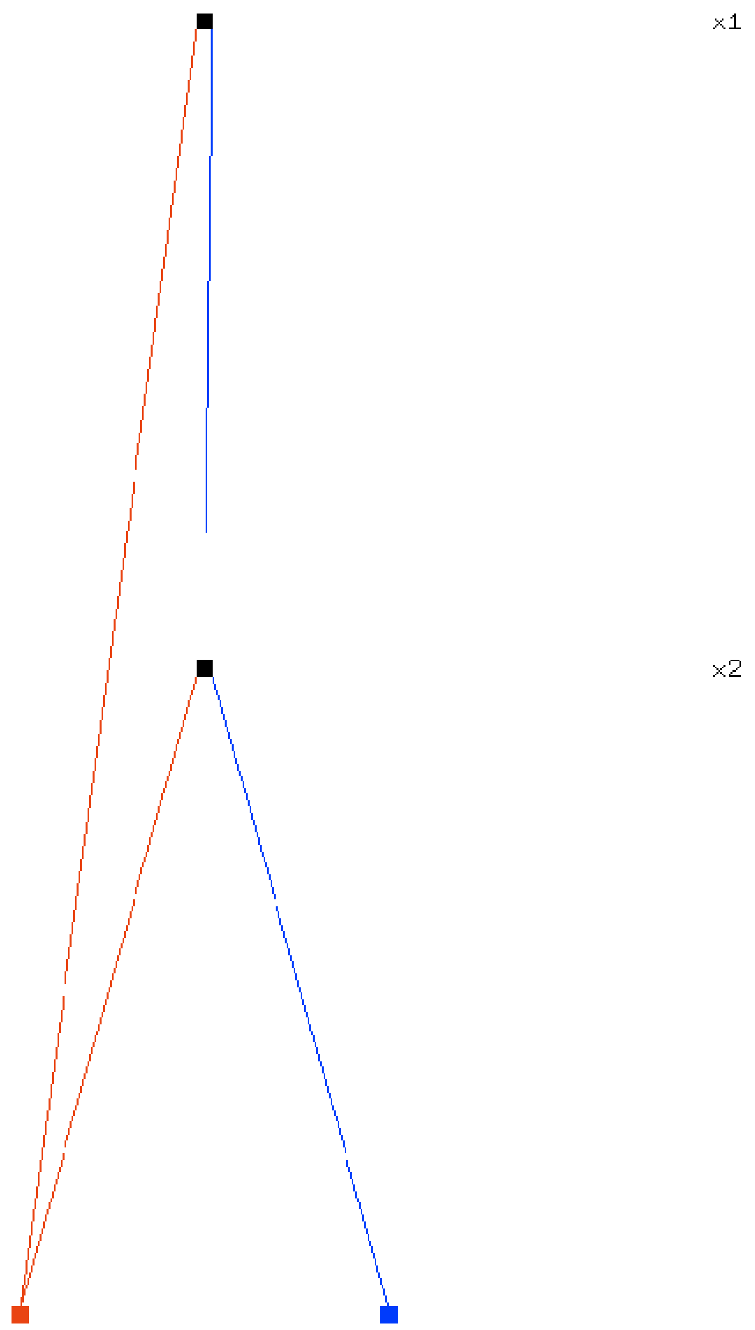


Figure 71. Symmetric Bent Function of Order 2 (x_1x_2).

B. SYMMETRIC BENT FUNCTION OF ORDER 4

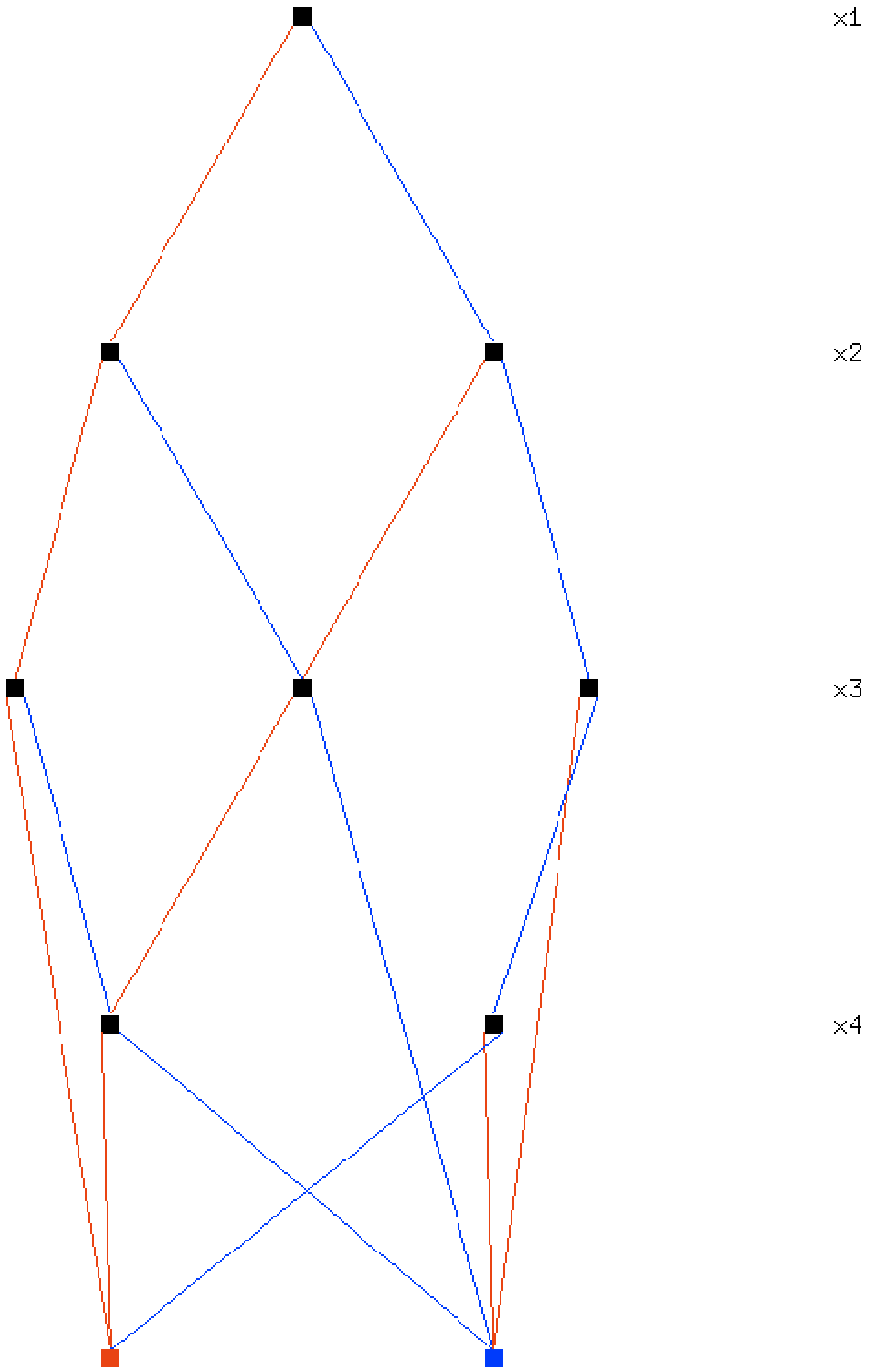


Figure 72. Symmetric Bent Function of Order 4.

C. SYMMETRIC BENT FUNCTION OF ORDER 6

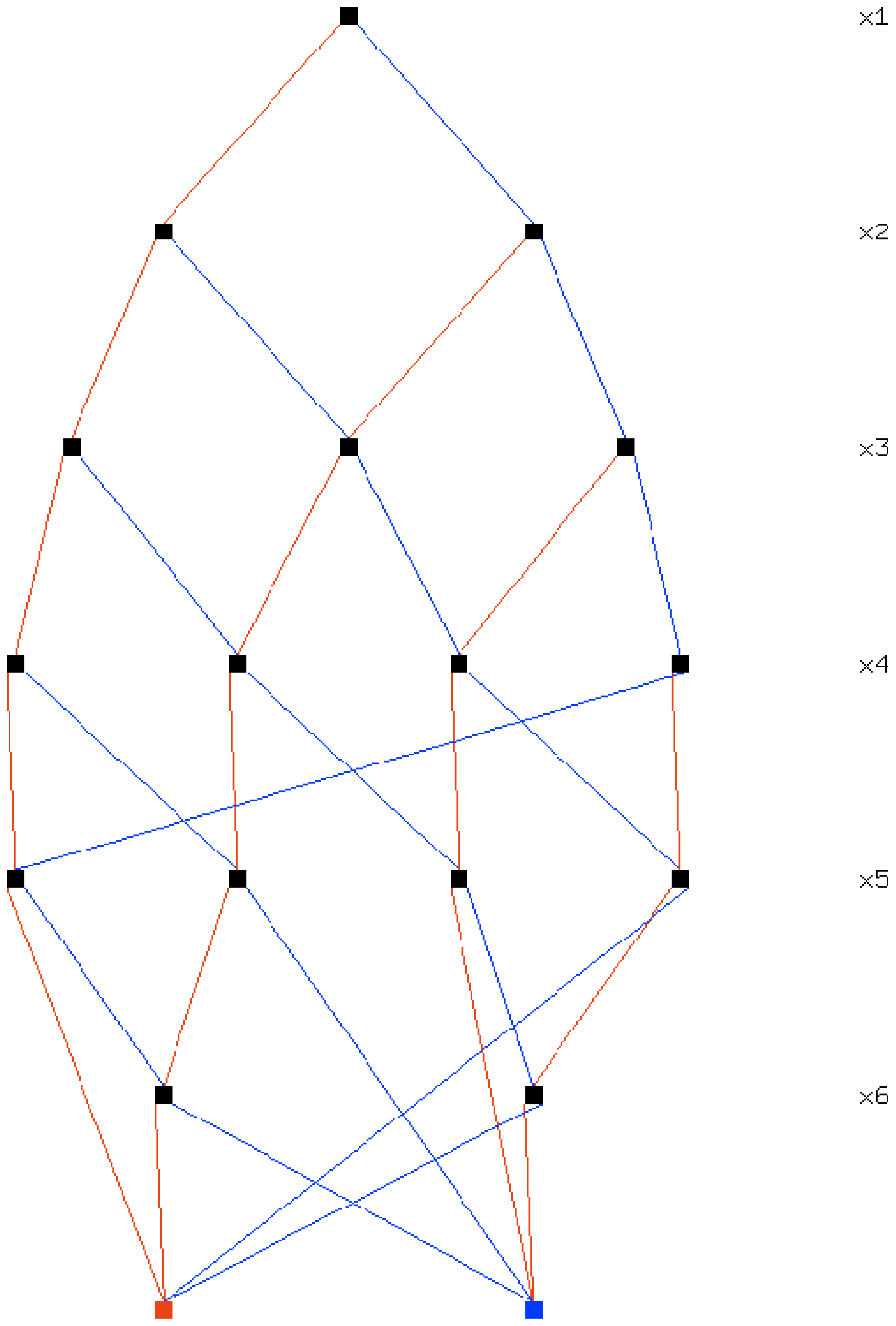


Figure 73. Symmetric Bent Function of Order 6

D. SYMMETRIC BENT FUNCTION OF ORDER 8

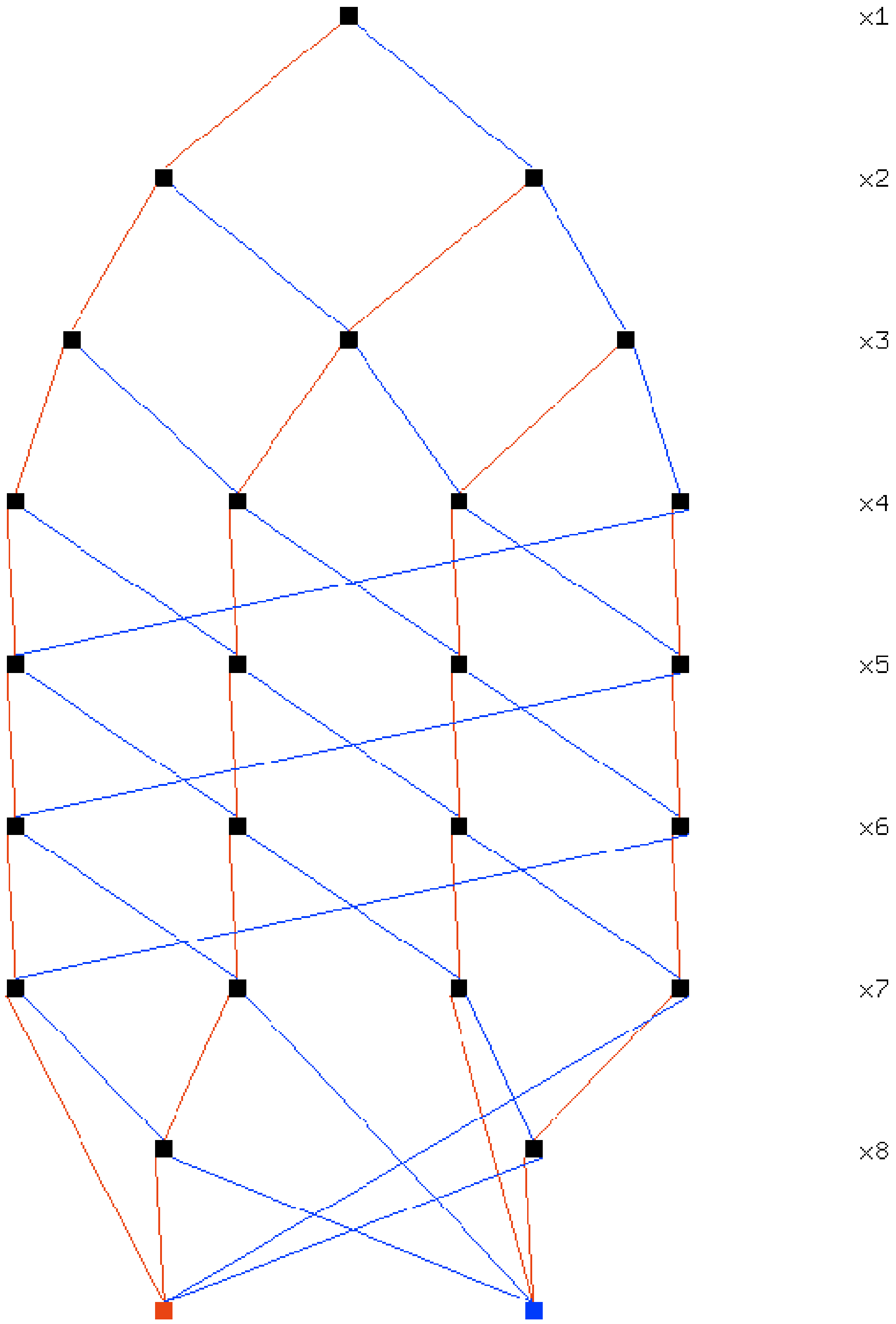


Figure 74. Symmetric Bent Function of Order 8

APPENDIX D: MISCELLANEOUS BDDS

A. PARTIAL SELECTION OF HOMOGENEOUS FUNCTIONS ON 8-VARIABLES OF DEGREE 3 (MINIMUMS BDDS)

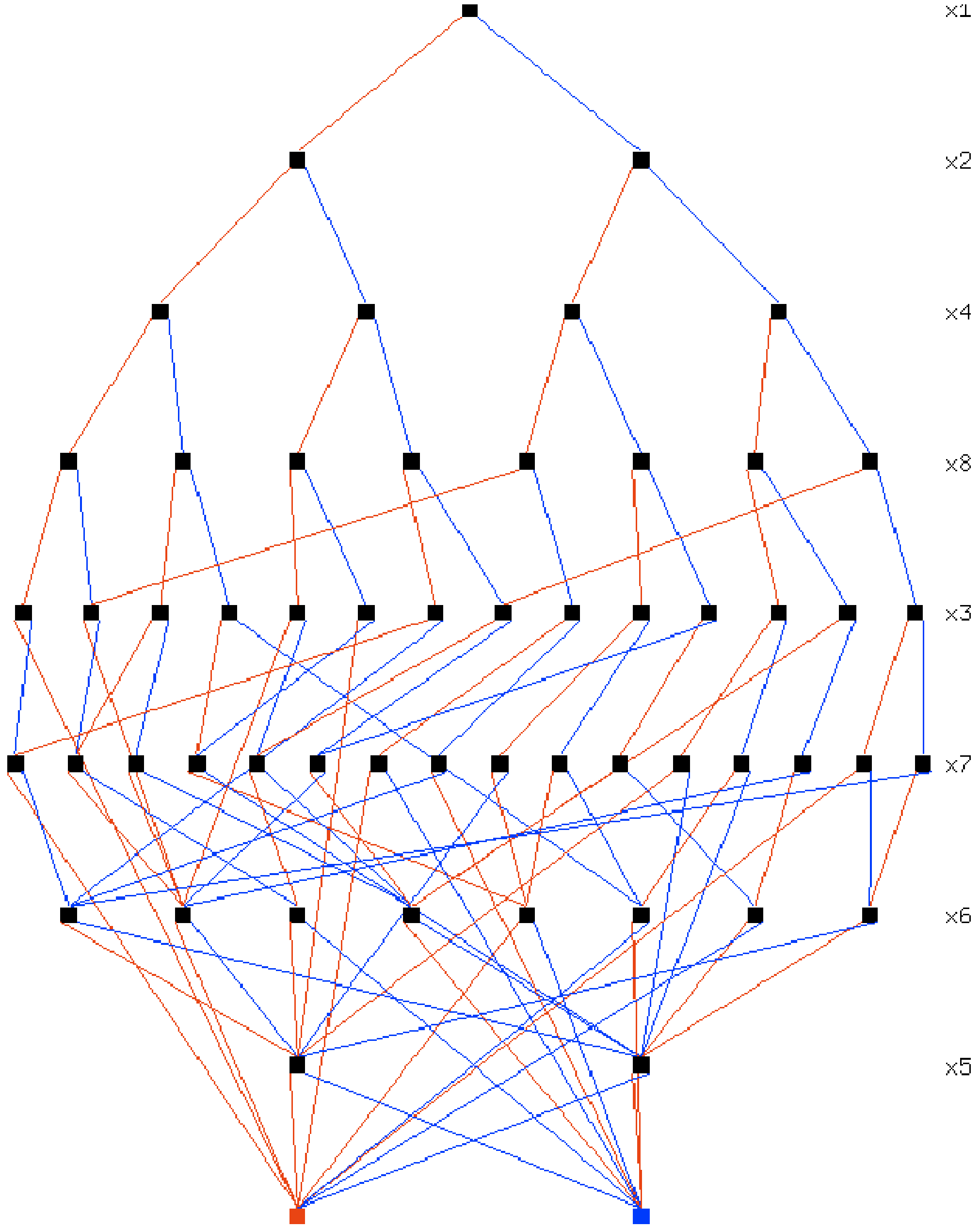


Figure 75. $x_1x_2x_3 \oplus x_1x_2x_5 \oplus x_1x_2x_7 \oplus x_1x_2x_8 \oplus x_1x_3x_8 \oplus x_1x_4x_6 \oplus x_1x_4x_8 \oplus x_1x_5x_6 \oplus x_1x_7x_8 \oplus x_2x_3x_5 \oplus x_2x_3x_6 \oplus x_2x_3x_7 \oplus x_2x_5x_6 \oplus x_2x_5x_8 \oplus x_3x_4x_7 \oplus x_3x_4x_8 \oplus x_3x_5x_7 \oplus x_3x_6x_7 \oplus x_4x_5x_6 \oplus x_4x_5x_7 \oplus x_4x_6x_7 \oplus x_4x_6x_8 \oplus x_4x_7x_8 \oplus x_5x_6x_8$

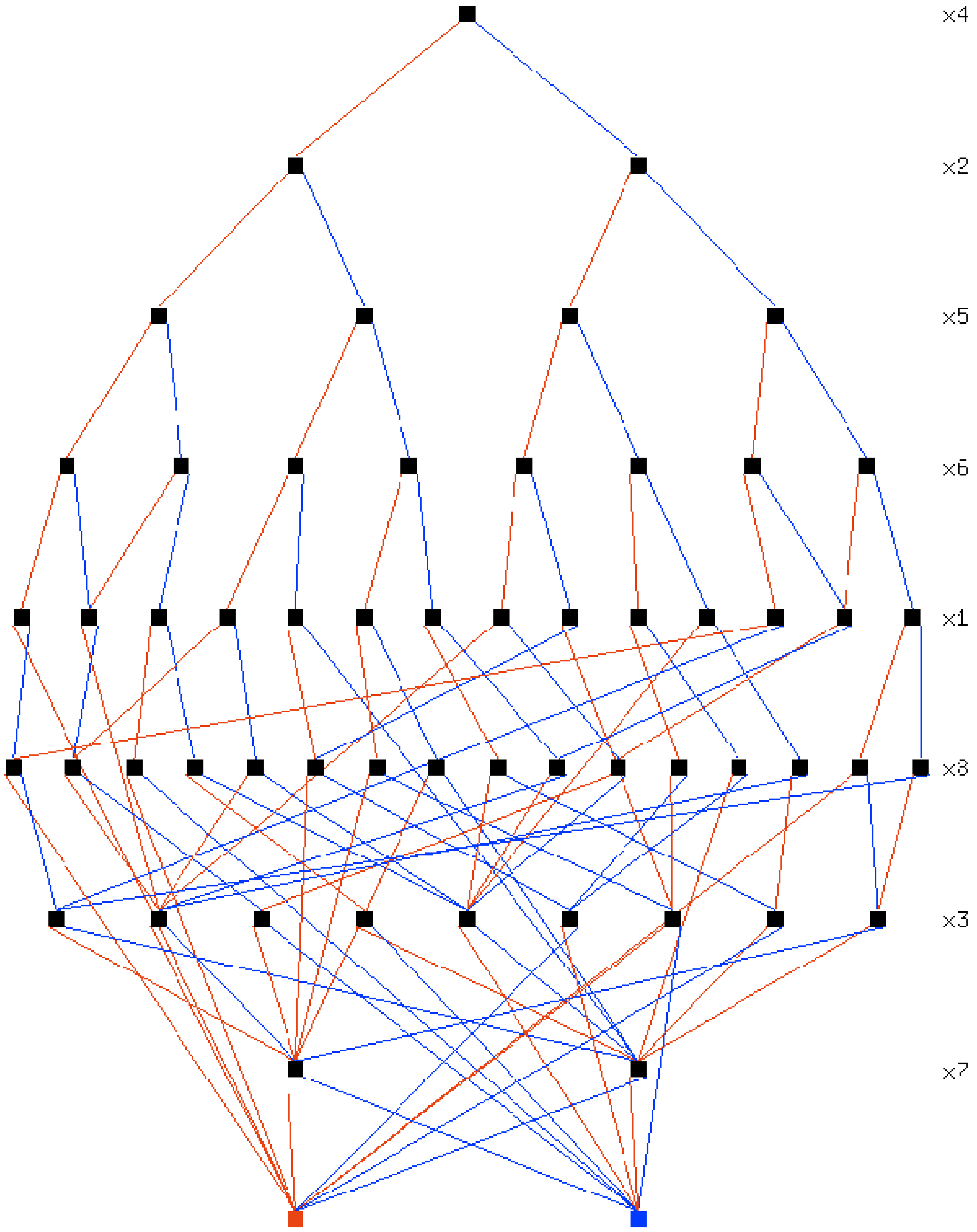


Figure 76. $x_1x_2x_6 \oplus x_1x_2x_8 \oplus x_1x_3x_4 \oplus x_1x_3x_8 \oplus x_1x_4x_5 \oplus x_1x_4x_7 \oplus x_1x_4x_8 \oplus x_1x_5x_6 \oplus x_1x_7x_8 \oplus x_2x_3x_7 \oplus x_2x_3x_8 \oplus x_2x_5x_6 \oplus x_2x_5x_7 \oplus x_2x_6x_7 \oplus x_2x_6x_8 \oplus x_2x_7x_8 \oplus x_3x_4x_5 \oplus x_3x_4x_6 \oplus x_3x_4x_7 \oplus x_3x_5x_7 \oplus x_3x_6x_7 \oplus x_4x_5x_6 \oplus x_4x_5x_8 \oplus x_5x_6x_8$

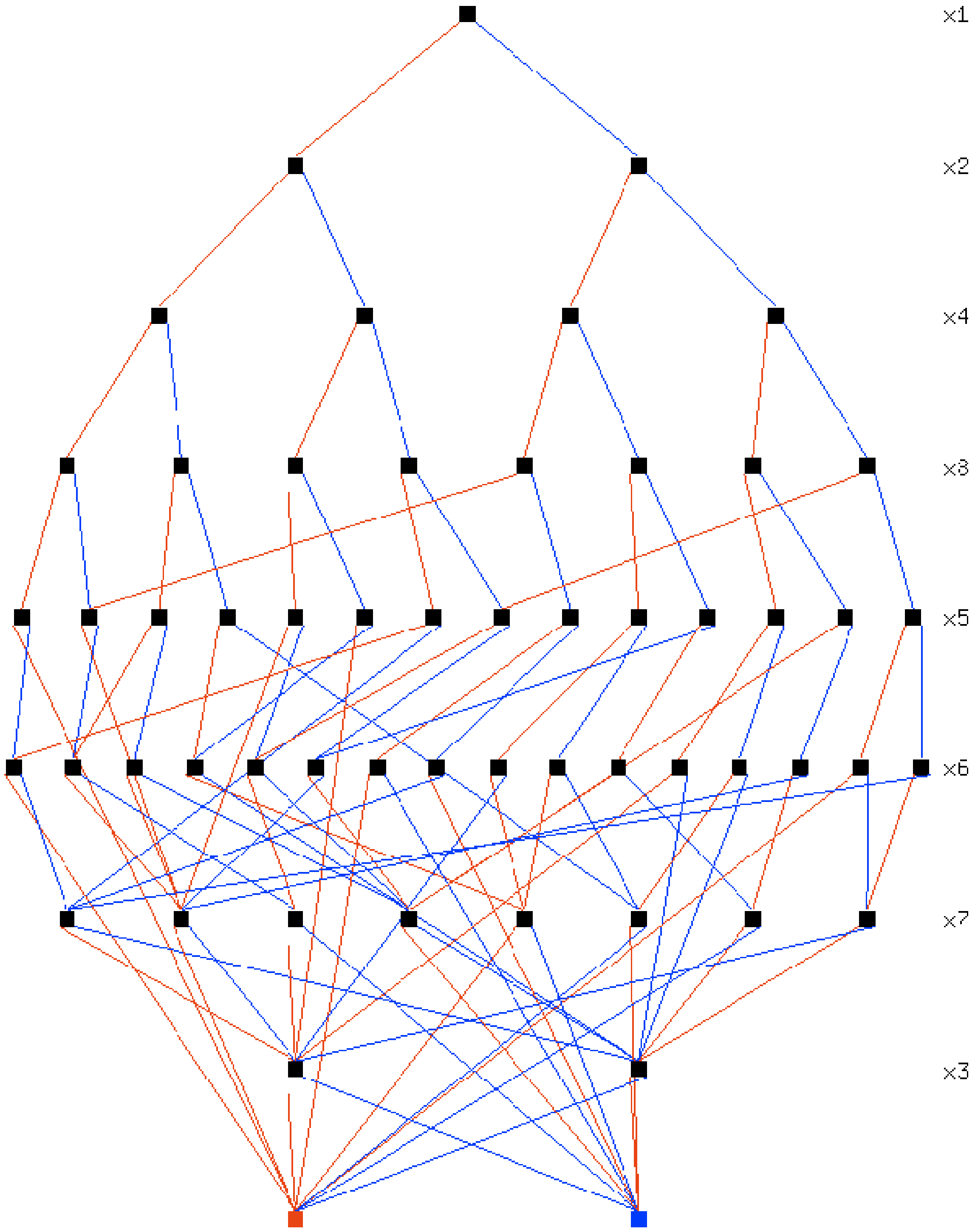


Figure 77. $x_1x_2x_3 \oplus x_1x_2x_5 \oplus x_1x_2x_6 \oplus x_1x_2x_8 \oplus x_1x_3x_5 \oplus x_1x_3x_6 \oplus x_1x_3x_7 \oplus x_1x_4x_7 \oplus$
$x_1x_4x_8 \oplus x_1x_5x_7 \oplus x_1x_5x_8 \oplus x_1x_6x_7 \oplus x_1x_6x_8 \oplus x_2x_3x_5 \oplus x_2x_3x_7 \oplus x_2x_3x_8 \oplus$
$x_2x_5x_6 \oplus x_2x_5x_7 \oplus x_3x_4x_6 \oplus x_3x_4x_7 \oplus x_3x_5x_6 \oplus x_3x_5x_8 \oplus x_3x_6x_8 \oplus x_3x_7x_8 \oplus$
$x_4x_5x_6 \oplus x_4x_5x_8 \oplus x_4x_6x_7 \oplus x_4x_6x_8 \oplus x_4x_7x_8 \oplus x_5x_6x_7 \oplus x_5x_7x_8 \oplus x_6x_7x_8$

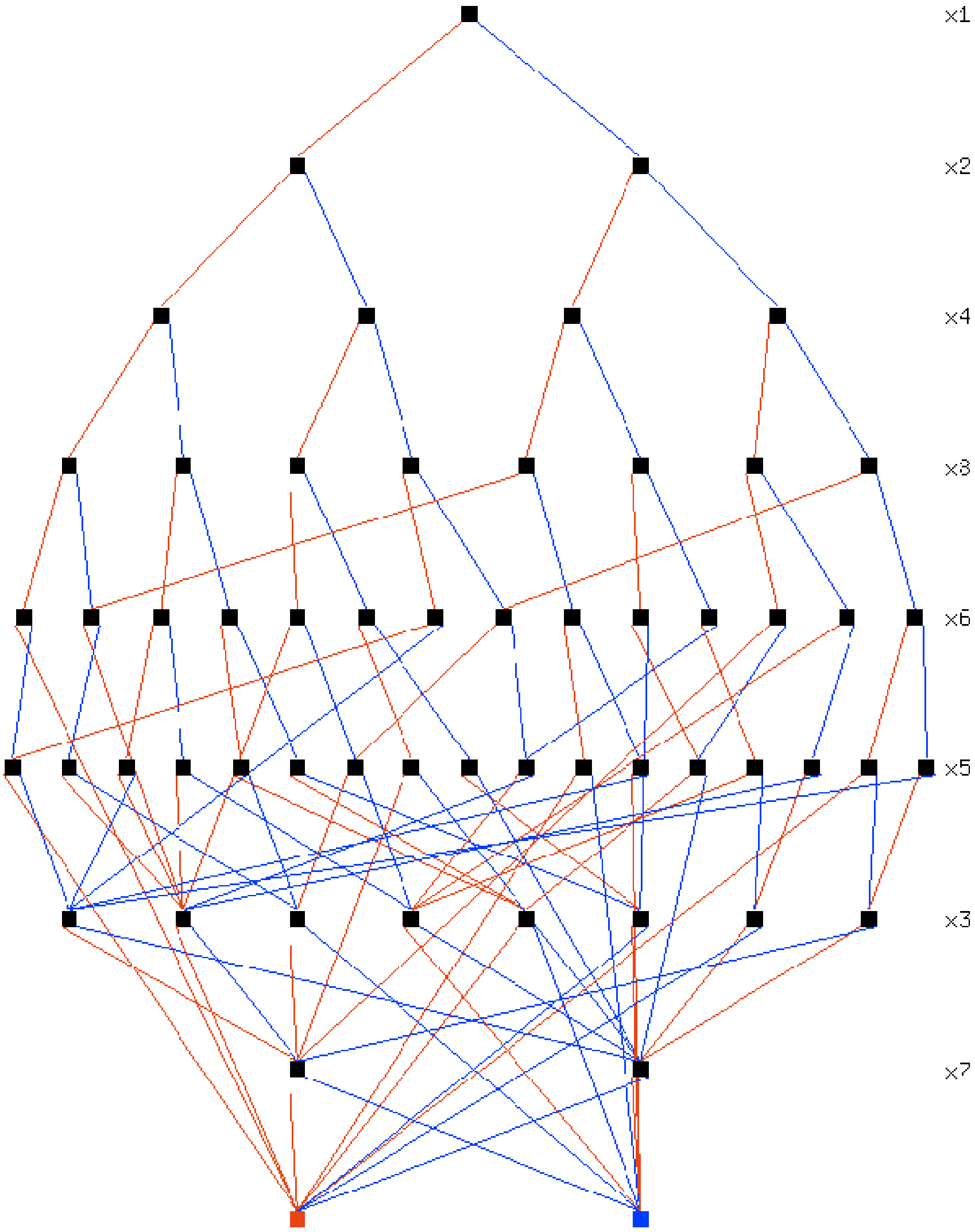


Figure 78. $x_1x_2x_3 \oplus x_1x_2x_5 \oplus x_1x_2x_6 \oplus x_1x_2x_8 \oplus x_1x_3x_7 \oplus x_1x_4x_7 \oplus x_1x_4x_8 \oplus x_1x_5x_8 \oplus x_1x_6x_8 \oplus x_2x_3x_5 \oplus x_2x_3x_7 \oplus x_2x_3x_8 \oplus x_2x_5x_6 \oplus x_2x_5x_7 \oplus x_3x_4x_6 \oplus x_3x_4x_7 \oplus x_3x_5x_6 \oplus x_3x_7x_8 \oplus x_4x_5x_6 \oplus x_4x_5x_8 \oplus x_4x_6x_7 \oplus x_4x_6x_8 \oplus x_4x_7x_8 \oplus x_5x_6x_7$

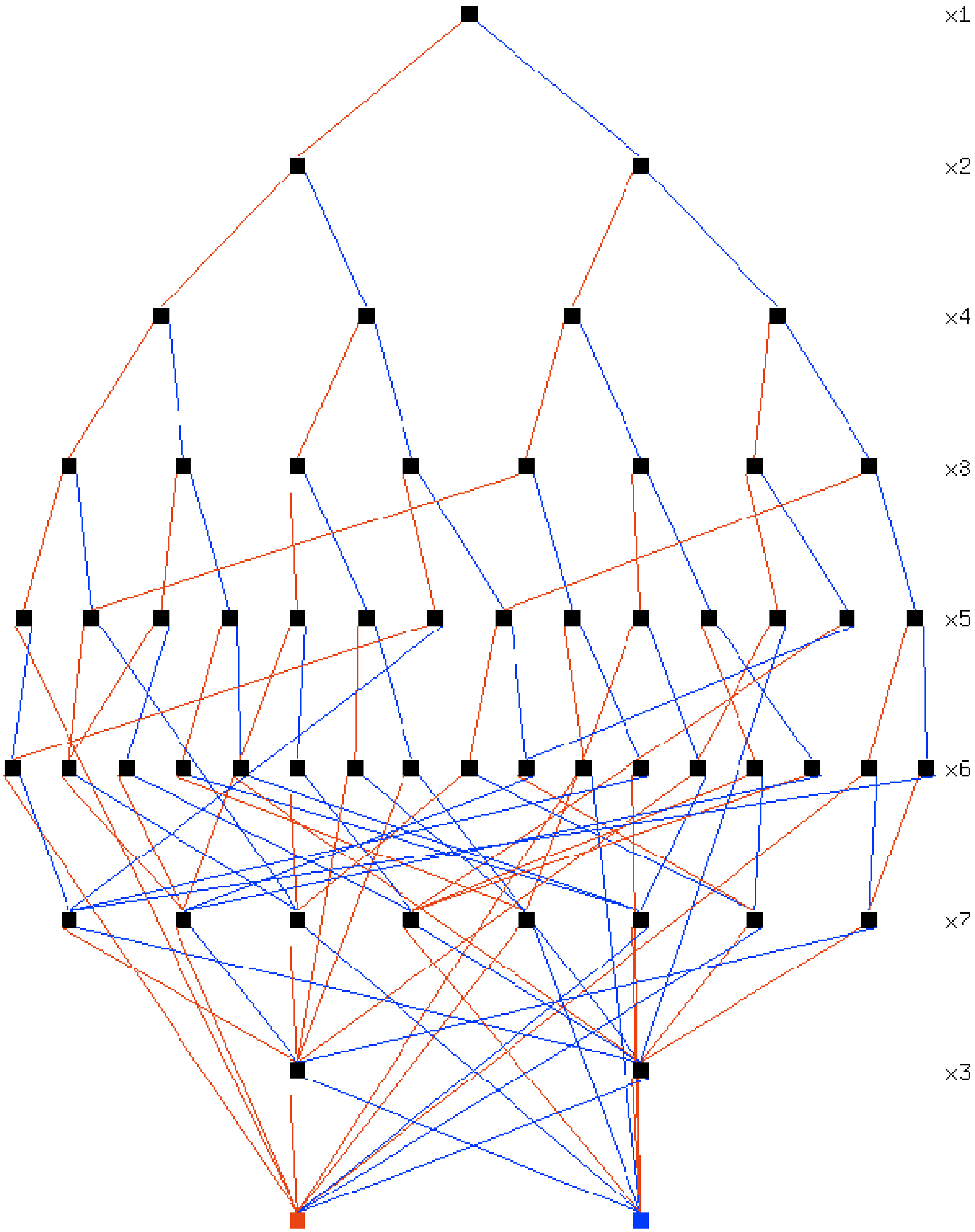


Figure 79. $x_1x_2x_7 \oplus x_1x_2x_8 \oplus x_1x_3x_4 \oplus x_1x_3x_7 \oplus x_1x_4x_5 \oplus x_1x_4x_6 \oplus x_1x_4x_8 \oplus x_1x_5x_8 \oplus x_1x_6x_8 \oplus x_2x_3x_6 \oplus x_2x_3x_7 \oplus x_2x_5x_6 \oplus x_2x_5x_8 \oplus x_2x_6x_7 \oplus x_2x_6x_8 \oplus x_2x_7x_8 \oplus x_3x_4x_5 \oplus x_3x_4x_7 \oplus x_3x_4x_8 \oplus x_3x_5x_6 \oplus x_3x_7x_8 \oplus x_4x_5x_6 \oplus x_4x_5x_7 \oplus x_5x_6x_7$

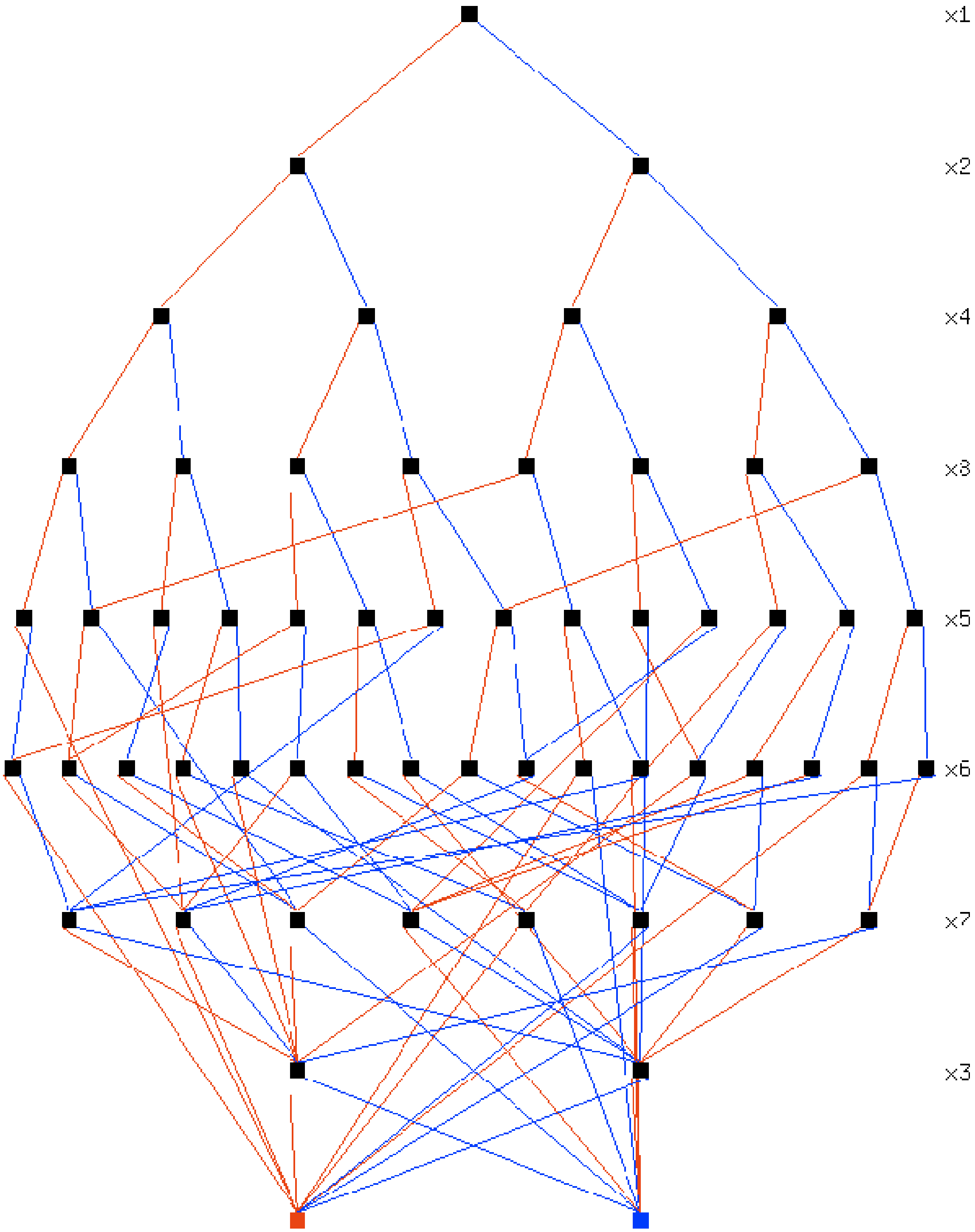


Figure 80.

$$x_1x_2x_7 \oplus x_1x_2x_8 \oplus x_1x_3x_4 \oplus x_1x_3x_5 \oplus x_1x_3x_6 \oplus x_1x_3x_7 \oplus x_1x_4x_5 \oplus x_1x_4x_6 \oplus x_1x_4x_8 \oplus x_1x_5x_7 \oplus x_1x_5x_8 \oplus x_1x_6x_7 \oplus x_1x_6x_8 \oplus x_2x_3x_6 \oplus x_2x_3x_7 \oplus x_2x_5x_6 \oplus x_2x_5x_8 \oplus x_2x_6x_7 \oplus x_2x_6x_8 \oplus x_2x_7x_8 \oplus x_3x_4x_5 \oplus x_3x_4x_7 \oplus x_3x_4x_8 \oplus x_3x_5x_6 \oplus x_3x_5x_8 \oplus x_3x_6x_8 \oplus x_3x_7x_8 \oplus x_4x_5x_6 \oplus x_4x_5x_7 \oplus x_5x_6x_7 \oplus x_5x_7x_8 \oplus x_6x_7x_8$$

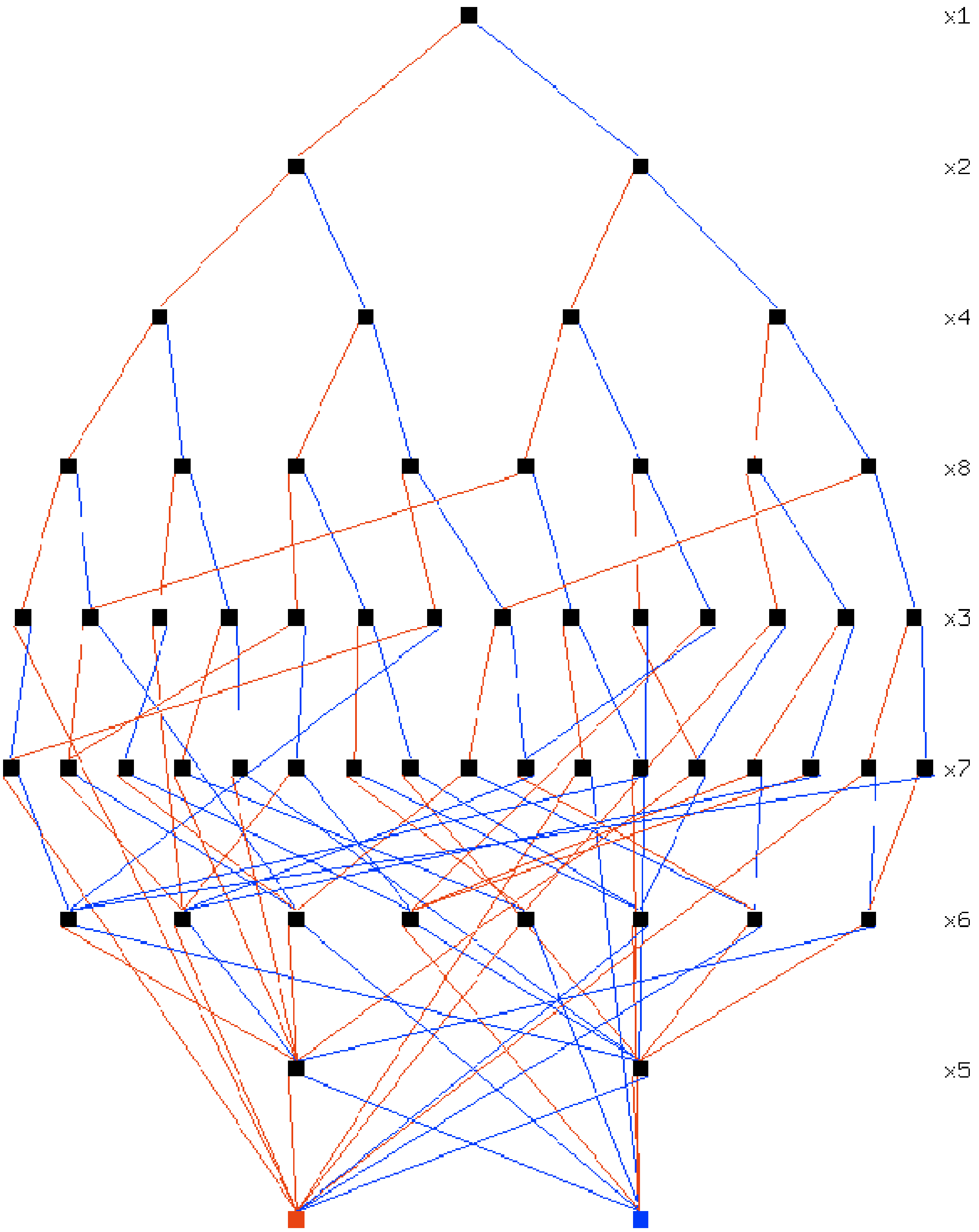


Figure 81. $x_1x_2x_6 \oplus x_1x_2x_8 \oplus x_1x_3x_4 \oplus x_1x_3x_5 \oplus x_1x_3x_6 \oplus x_1x_3x_8 \oplus x_1x_4x_5 \oplus x_1x_4x_7 \oplus x_1x_4x_8 \oplus x_1x_5x_6 \oplus x_1x_5x_7 \oplus x_1x_6x_7 \oplus x_1x_7x_8 \oplus x_2x_3x_7 \oplus x_2x_3x_8 \oplus x_2x_5x_6 \oplus x_2x_5x_7 \oplus x_2x_6x_7 \oplus x_2x_6x_8 \oplus x_2x_7x_8 \oplus x_3x_4x_5 \oplus x_3x_4x_6 \oplus x_3x_4x_7 \oplus x_3x_5x_7 \oplus x_3x_5x_8 \oplus x_3x_6x_7 \oplus x_3x_6x_8 \oplus x_4x_5x_6 \oplus x_4x_5x_8 \oplus x_5x_6x_8 \oplus x_5x_7x_8 \oplus x_6x_7x_8$

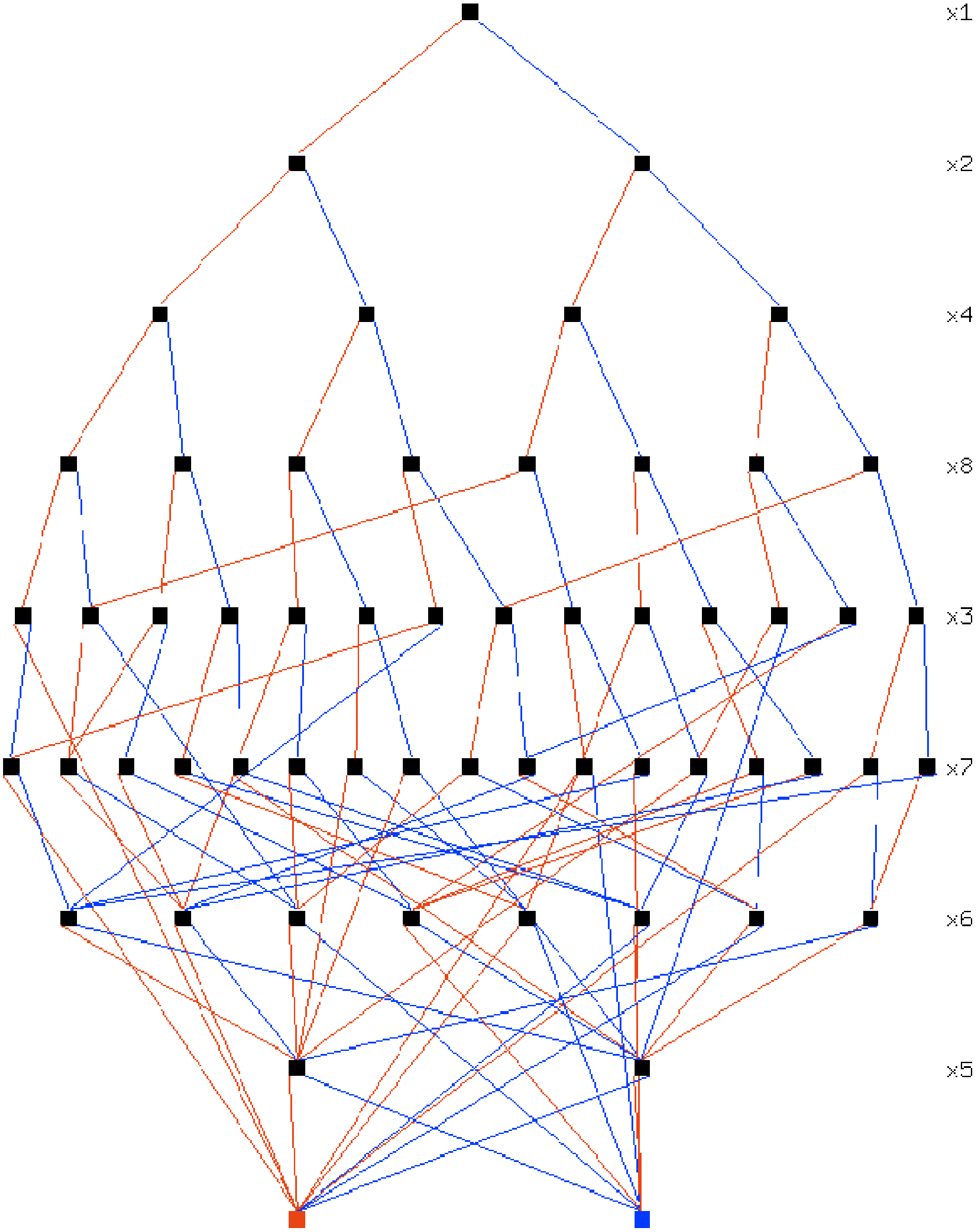


Figure 82. $x_1x_2x_3 \oplus x_1x_2x_5 \oplus x_1x_2x_7 \oplus x_1x_2x_8 \oplus x_1x_3x_5 \oplus x_1x_3x_6 \oplus x_1x_3x_8 \oplus x_1x_4x_6 \oplus x_1x_4x_8 \oplus x_1x_5x_6 \oplus x_1x_5x_7 \oplus x_1x_6x_7 \oplus x_1x_7x_8 \oplus x_2x_3x_5 \oplus x_2x_3x_6 \oplus x_2x_3x_7 \oplus x_2x_5x_6 \oplus x_2x_5x_8 \oplus x_3x_4x_7 \oplus x_3x_4x_8 \oplus x_3x_5x_7 \oplus x_3x_5x_8 \oplus x_3x_6x_7 \oplus x_3x_6x_8 \oplus x_4x_5x_6 \oplus x_4x_5x_7 \oplus x_4x_6x_7 \oplus x_4x_6x_8 \oplus x_4x_7x_8 \oplus x_5x_6x_8 \oplus x_5x_7x_8 \oplus x_6x_7x_8$

B. MISCELLANEOUS 8-VARIABLE BENT FUNCTIONS (MINIMUM BDDS)

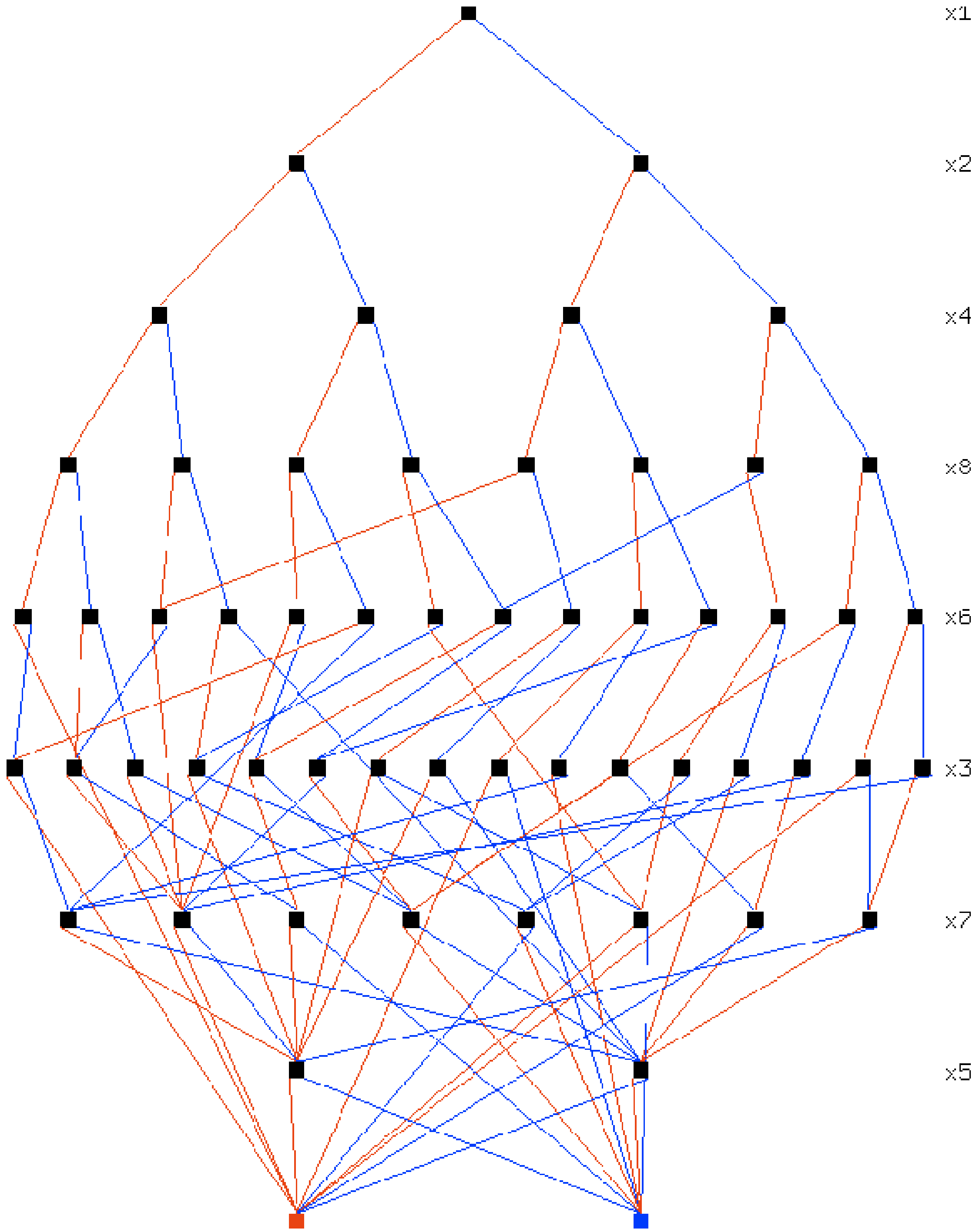


Figure 83. Hex Truth Table: 00110572175C476A 032E357E1B6C7869 00775F4E173AE2A9 3F74AC81D8C9E196

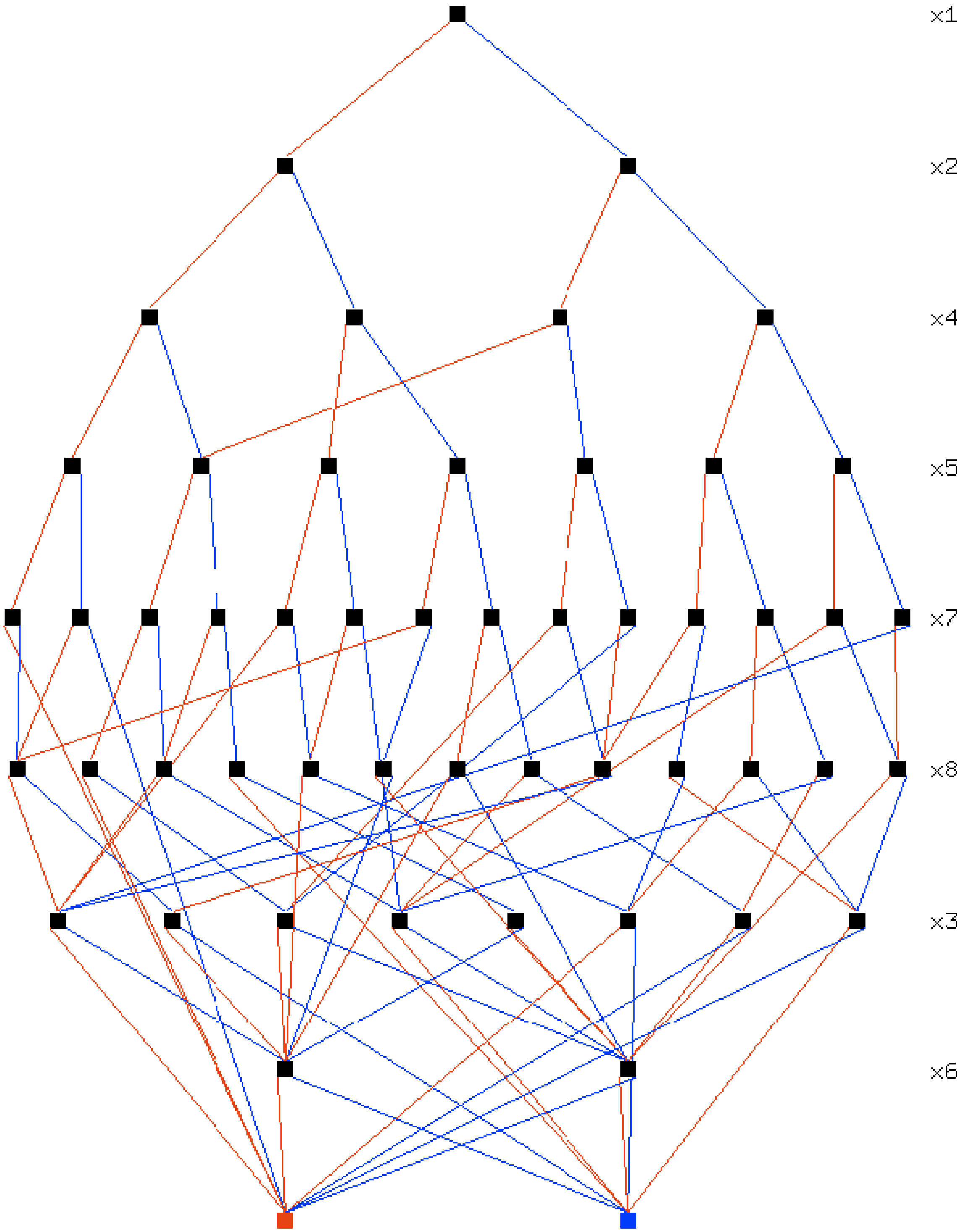


Figure 84. Hex Truth Table: 01041576134C526B 023B257A1F7C6D68 15760E0B526BE3BC 2A75FDC49D98E083

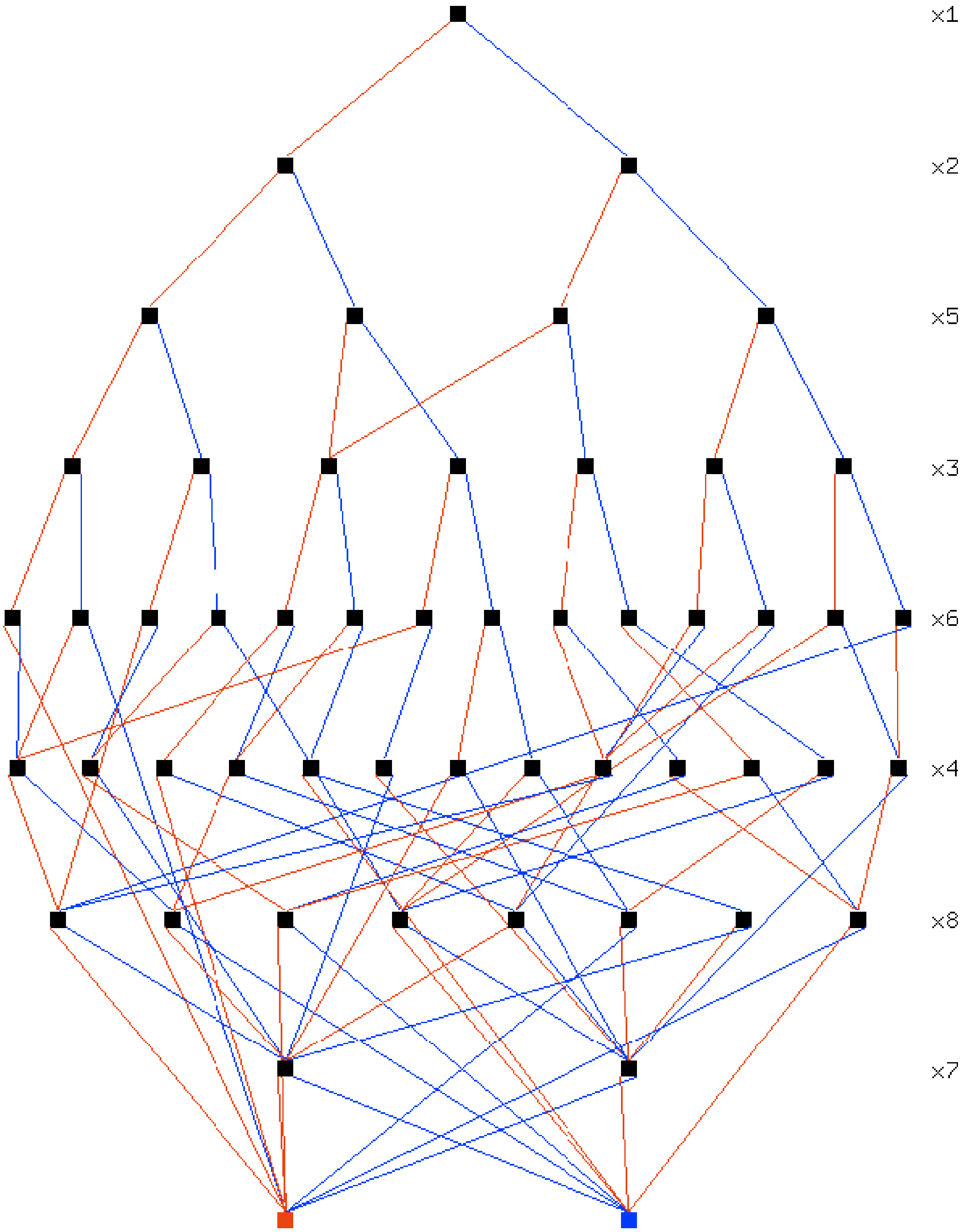


Figure 85. Hex Truth Table: 01150713105E703E 071C68737F3E89C8 077A68157F5889AE 67EA61EC76A116C1

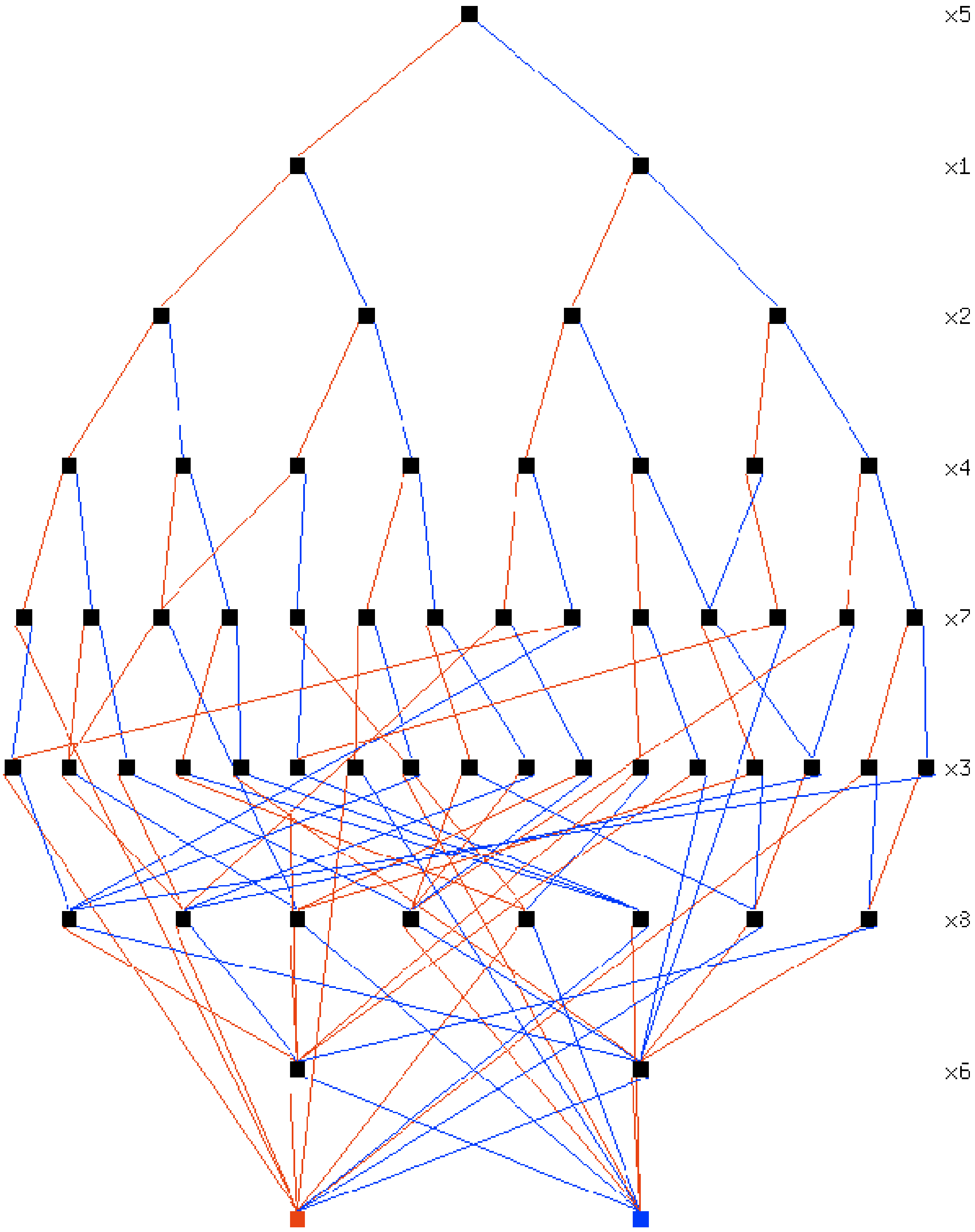


Figure 86. Hex Truth Table: 0017051212367E5A 170F746C5F74AA9 1173C476C5FB8668 133E8FA21DEC98196

LIST OF REFERENCES

[1] O. S. Rothaus, “On ‘Bent’ Functions,” *Journal of Combinatorial Theory*, ser. A, vol. 20, pp. 300–305, 1976.

[2] T. W. Cusick and P. Stanica, *Cryptographic Boolean Functions and Application.* San Diego: Academic Press, 2009.

[3] T. Sasao, H. Nakahara, M. Matsuura, Y. Kawamura, H. Kajiwara, and J. T. Butler, “A Quaternary Decision Diagram Machine and the Optimization of Its Code,” presented at the International Symposium on Multiple Valued Logic, Naha, Okinawa, Japan, 2009, pp. 362–369.

[4] C. E. Shannon, “Communication Theory of Secrecy Systems,” *Bell Systems Technical Journal*, vol. 28, pp. 656–715, 1949.

[5] K. Goossens, “Automated Creation and Selection of Cryptographic Primitives,” M.S. thesis, Katholieke Universiteit Leuven, Leuven, Belgium, 2006.

[6] M. Matsui and A. Yamagishi, “A New Method for Known Plaintext Attack of FEAL Cipher,” *Advances in Cryptology – Eurocrypt ’92*, Berlin: Spring, 1993.

[7] J. T. Butler and T. Sasao, “Logic Functions for Cryptography – A Tutorial,” in *Proceedings of the Reed-Muller Workshop*, 2009, pp. 127–136.

[8] B. Preneel, “Analysis and Design of Cryptographic Hash Functions,” Ph.D. dissertation, Katholieke Universiteit Leuven, Leuven, Belgium, 1993.

[9] C. Carlet and A. Klapper, “Upper Bounds on the Numbers of Resilient Functions and of Bent Functions,” in *Proceedings of the 23rd Symposium on Information Theory in the Benelux*, 2002, pp. 307–314.

[10] P. Langevin, “Classification of Boolean Quartics Forms in Eight Variables,” July 2008 [Online]. Available: http://langevin.univ-tln.fr/project/quartics/. [Accessed: Aug. 28, 2009].

[11] J. F. Dillon, “A Survey of Bent Functions,” *NSA Technical Journal*, Special Issue, pp. 191–215, 1972.

[12] P. Savicky, “On Bent Functions that are Symmetric,” *European Journal of Combinatorials*, vol. 15, pp. 407–410, 1994.

[13] R. E. Bryant, “Symbolic Boolean Manipulation with Ordered Binary Decision Diagrams,” *ACM Computing Surveys*, vol. 24, no. 23, September, 1992, pp. 293–318.

[14] J.-F. Michon, J.-B. Yunes, and P. Valarcher, "On Maximal Boolean Complexity Functions," *Theoretical Informatics and Applications*, vol. 39, 2005, pp. 677–689.

[15] T. Sasao and J. T. Butler, "On the Size of BDD Programs," Preprint, 2009.

[16] J.T. Butler, D.S. Herscovici, T. Sasao, and R.J. Barton III, "Average and Worst Case Number of Nodes in Decision Diagrams of Symmetric Multiple-Valued Functions," *IEEE Transactions on Computers*, vol.46, no. 4, April 1997, pp. 491–495.

[17] J.T. Butler, P. Stanica, N. B. Schafer, "Properties of the BDDs of Bent Functions," Unpublished.

[18] W. Gunther and R. Drechsler, "BDD Minimization by Linear Transformations," in International Conference on Advanced Computer Systems No. 5, 1998, pp. 525–532.

[19] F. Somenzi, "CUDD: CU Decision Diagram Package," February 2009 [Online]. Available: http://vlsi.colorado.edu/~fabio/CUDD/ [Accessed: Aug. 25, 2009].

[20] A. Bilgin, et al, *Graphviz.* [Download]. Florham Park, NJ: AT&T Research, 2009.

[21] H. F. Trotter, "Perm (Algorithm 115)," *Communications of the ACM*, Ser. 5, Vol. 8, pp. 434–435, August 1962.

[22] S. M. Johnson, "Generation of Permutations by Adjacent Transposition," *Mathematics of Computation*, Ser. 17, Vol. 83, pp. 282–285, July 1963.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
Ft. Belvoir, Virginia

2. Dudley Knox Library
Naval Postgraduate School
Monterey, California

3. Jon T. Butler
Naval Postgraduate School
Monterey, California

4. Pantelimon Stanica
Naval Postgraduate School
Monterey, California

5. John G. Harkins
National Security Agency
Ft. Meade, Maryland

6. David R. Podany
National Security Agency
Ft. Meade, Maryland